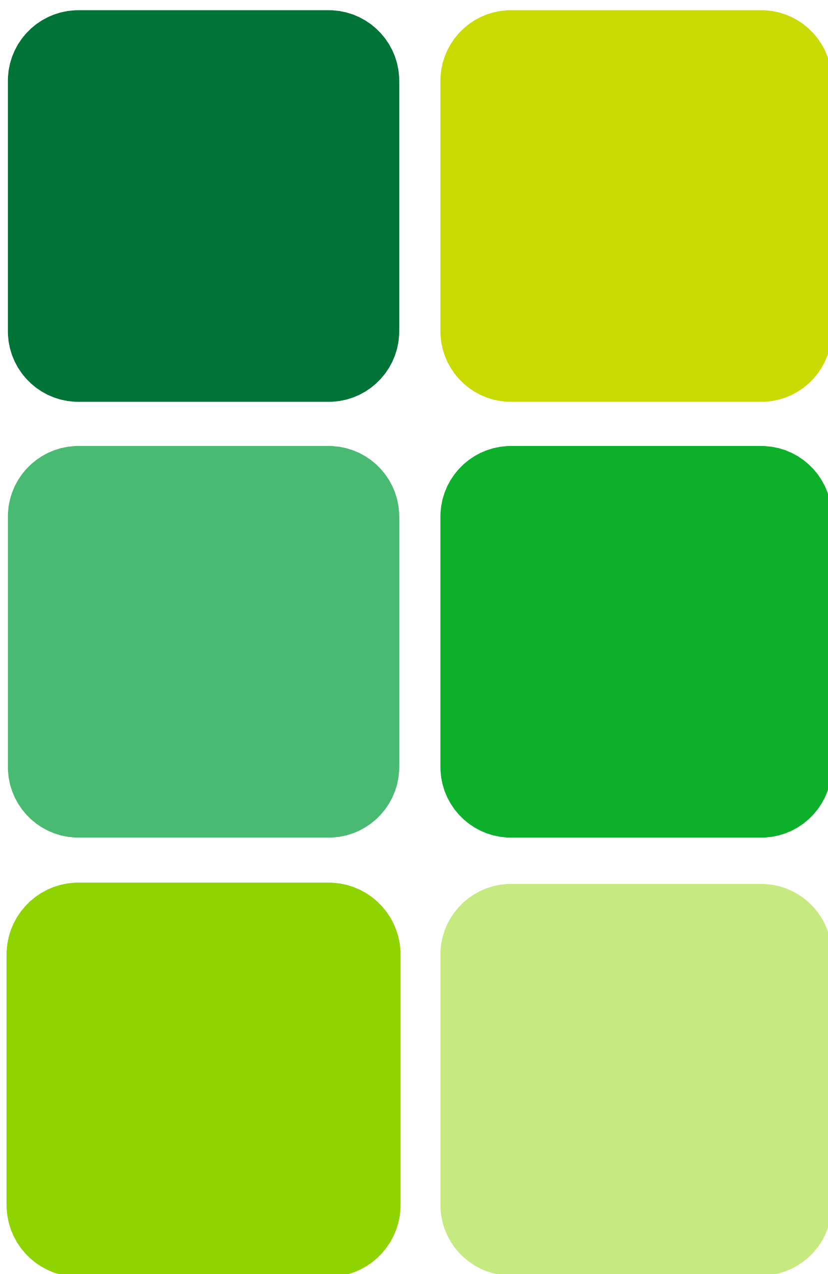


www.sapientia.uji.es | 93

UNIVERSITAT JAUME•I

Introducción a la programación con Python 3

Andrés Marzal Varó
Isabel Gracia Luengo
Pedro García Sevilla

Introducción a la programación con Python 3

Andrés Marzal Varó
Isabel Gracia Luengo
Pedro García Sevilla



UNIVERSITAT
JAUME • I

Departament de Llenguatges i Sistemes Informàtics

■ Codis d'assignatures EI1003 i MT1003

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93



Edita: Publicacions de la Universitat Jaume I. Servei de Comunicació i Publicacions
Campus del Riu Sec. Edifici Rectorat i Serveis Centrals. 12071 Castelló de la Plana
http://www.tenda.uji.es e-mail: publicacions@uji.es

Col·lecció Sapientia 93
www.sapientia.uji.es
Primera edició, 2014

ISBN: 978-84-697-1178-1



Publicacions de la Universitat Jaume I és una editorial membre de l'UNE, cosa que en garanteix la difusió de les obres en els àmbits nacional i internacional. www.une.es



Índice general

Prefacio

«Introducción a la programación con Python 3» desarrolla el temario de la asignatura «Programación I» que se imparte durante el primer semestre de primer curso en los grados en Ingeniería Informática y en Matemática Computacional de la Universitat Jaume I. En ella se pretende enseñar a programar y se utiliza Python como primer lenguaje de programación.

¿Por qué Python? Python es un lenguaje de muy alto nivel que permite expresar algoritmos de forma casi directa (ha llegado a considerarse «pseudocódigo ejecutable») y hemos comprobado que se trata de un lenguaje particularmente adecuado para la enseñanza de la programación. Esta impresión se ve corroborada por la adopción de Python como lenguaje introductorio en otras universidades. Otros lenguajes, como Java, C o C#, exigen una gran atención a multitud de detalles que dificultan la implementación de algoritmos a un estudiante que se enfrenta por primera vez al desarrollo de programas. No obstante, son lenguajes de programación de referencia y deberían formar parte del currículum de todo informático. Aprender Python como primer lenguaje permite estudiar las estructuras de control y de datos básicas con un alto nivel de abstracción y, así, entender mejor qué supone exactamente la mayor complejidad de la programación en otros lenguajes y hasta qué punto es mayor el grado de control que nos otorgan. Por ejemplo, una vez se han estudiado listas en Python, su implementación en otros lenguajes permite al estudiante no perder de vista el objetivo último: construir una entidad con cierto nivel de abstracción usando las herramientas concretas proporcionadas por el lenguaje. De algún modo, pues, Python ayuda al aprendizaje posterior de otros lenguajes, lo que proporciona al estudiante una visión más rica y completa de la programación. Las similitudes y diferencias entre los distintos lenguajes permiten al estudiante inferir más fácilmente qué es fundamental y qué accesorio o accidental al diseñar programas en un lenguaje de programación cualquiera.

¿Y por qué otro libro de texto introductorio a la programación? Ciertamente hay muchos libros que enseñan a programar desde cero. Creemos que este texto se diferencia de ellos en la forma en que se exponen y desarrollan los contenidos. Hemos procurado adoptar siempre el punto de vista del estudiante y presentar los conceptos y estrategias para diseñar programas básicos paso a paso, incrementalmente. La experiencia docente nos ha ido mostrando toda una serie de líneas de razonamiento inapropiadas, errores y vicios en los que caen muchos estudiantes. El texto intenta exponer, con mayor o menor fortuna, esos razonamientos, errores y vicios para que el estudiante los tenga presentes y procure evitarlos. Así, en el desarrollo de algunos programas llegamos a ofrecer versiones erróneas para, acto seguido, estudiar sus defectos y mostrar una versión corregida. El libro está repleto de cuadros que pretenden profundizar en aspectos marginales, llamar la atención sobre algún extremo, ofrecer algunas pinceladas de historia o, sencillamente, desviarse de lo sustancial con alguna digresión que podría resultar motivadora para el estudiante.

Queremos aprovechar para dar un consejo al estudiantado que no nos cansamos de repetir: es *imposible* aprender a programar limitándose a leer un texto o a seguir pasivamente una explicación en clase (especialmente si el período de estudio se concentra en una o dos semanas antes del examen). Programar al nivel propio de un curso introductorio no es particularmente difícil, pero constituye una actividad intelectual radicalmente nueva para muchos estudiantes. Es necesario darse una oportunidad para ir asentando los conocimientos y las estrategias de diseño de programas (y así, superar el curso). Esa oportunidad requiere tiempo para madurar… y trabajo, mucho trabajo; por eso el texto ofrece más de cuatrocientos ejercicios. Solo tras haberse enfrentado a buena parte de ellos se estará preparado para demostrar que se ha aprendido lo necesario.

Este texto es una actualización y revisión del libro *Introducción a la programación con Python* publicado dentro de la colección Sapientia de la Universitat Jaume I. Las principales diferencias con respecto a la versión anterior son:

- Se ha actualizado la versión del lenguaje empleado. La versión anterior del libro se escribió para Python 2 (en concreto, para la versión 2.3). Desde entonces, el lenguaje ha seguido dos ramas de desarrollo diferentes: Python 2 y Python 3. Este nuevo texto utiliza Python 3, lo que, además de modificar contenidos, ha supuesto reescribir «todos» los programas de ejemplo. Aunque en el momento de escribir estas líneas ya se ha publicado la versión 3.4 de Python, todos los programas de ejemplo incluidos en este libro deberían funcionar para cualquier versión a partir de la 3.1.
- También hemos cambiado el entorno de desarrollo y la librería gráfica empleada. La versión anterior del libro usaba un entorno de desarrollo propio (PythonG) que, a su vez, incorporaba una sencilla librería gráfica. Ahora se utiliza un entorno de desarrollo estándar y muy potente, Eclipse, que podrá seguir siendo usado por los estudiantes en otras asignaturas y con otros lenguajes de programación. En cuanto a la parte gráfica, ahora se utiliza el módulo *turtle*, incorporado al propio lenguaje desde la versión 3.1.
- El capítulo 7 del libro anterior resultaba un tanto artificial al *simular*, mediante una librería propia, el concepto de «registro» como una versión simplificada del concepto de clase. Aunque la programación orientada a objetos se estudia en detalle en otras asignaturas de los grados, hemos considerado más acertado reescribir este capítulo para introducir los conceptos básicos de clase y objeto. Además, se ha incluido también un breve apartado dedicado al estudio de diccionarios.

Por supuesto, hemos corregido erratas (¡y también errores importantes!), hemos añadido nuevos ejemplos y modificado otros, hemos incorporado nuevos ejercicios y reubicado otros en lugares que hemos juzgado más apropiados, etc.

Convenios tipográficos

Hemos tratado de seguir una serie de convenios tipográficos a lo largo del texto. Los programas, por ejemplo, se muestran con fondo gris, así:

```
1 print('¡Hola,␣mundo!')
```

Por regla general, las líneas del programa aparecen numeradas a mano izquierda. Esta numeración tiene por objeto facilitar la referencia a puntos concretos del programa y no debe reproducirse en el fichero de texto si se copia el programa.

Cuando se quiere destacar el nombre del fichero en el que reside un programa, se indica en una barra encima del código:

```
hola_mundo.py
1 print('¡Hola,␣mundo!')
```

Cuando un programa contiene algún error grave, aparece un rayo rojo a la izquierda del nombre del programa:

```
ϟ hola_mundo.py
1 rint('¡Hola,␣mundo!')
```

Algunos programas no están completos y, por ello, presentan alguna deficiencia. No obstante, hemos optado por no marcarlos como erróneos cuando estos evolucionan en el curso de la exposición.

La información que aparece en pantalla se muestra con un tipo de letra monoespaciado. Así, el resultado de ejecutar la versión correcta del programa *hola_mundo.py* se mostrará como:

```
¡Hola, mundo!
```

Las sesiones interactivas del intérprete de Python también se muestran con un tipo de letra monoespaciado. El *prompt* primario del intérprete Python se muestra con los caracteres «>>>» y el secundario con «...».

Las expresiones y sentencias que debe teclear el usuario, tanto en el intérprete de Python como durante la ejecución de un programa, se destacan en color azul y el retorno de carro se representa explícitamente con el símbolo ↵ :

```
>>> '¡Hola,' + '␣' + 'mundo!'↵
'¡Hola,␣mundo!'
>>> if 'Hola' == 'mundo':↵
...     print('sí')↵
>>> else:↵
...    print('no')↵
... ↵
no
```

Agradecimientos

Este texto es fruto de la experiencia docente de todo el profesorado de las asignaturas «Metodología y tecnología de la programación» (de las antiguas titulaciones de Ingeniería Informática e Ingeniería Técnica en Informática de Gestión) y «Programación I» (de los actuales grados en Ingeniería Informática y en Matemática Computacional) de la Universitat Jaume I y se ha enriquecido con las aportaciones, comentarios y correcciones de muchos profesores del Departamento de Lenguajes y Sistemas Informáticos: Juan Pablo Aibar Ausina, Rafael Berlanga Llavori, Antonio Castellanos López, Víctor Manuel Jiménez Pelayo, María Dolores Llidó Escrivá, David Llorens Piñana, José Luis Llopis Borrás, Ramón Mollineda Cárdenas, Federico Prat Villar y Juan Miguel Vilar Torres. Para todos ellos, nuestro más sincero agradecimiento. Finalmente, también agradecemos la colaboración de cuantos nos han hecho llegar sugerencias o han detectado erratas.

Castellón de la Plana, 28 de julio de 2014
Andrés Marzal Varó, Isabel Gracia Luengo y Pedro García Sevilla

Capítulo 1

Introducción

—¿Qué sabes de este asunto?— preguntó el Rey a Alicia.
—Nada— dijo Alicia.
—¿*Absolutamente* nada?— insistió el Rey.
—Absolutamente nada— dijo Alicia.
—Esto es importante— dijo el Rey, volviéndose hacia los jurados.

Alicia en el país de las maravillas, Lewis Carroll

El objetivo de este curso es enseñarte a *programar*, esto es, a diseñar *algoritmos* y expresarlos como *programas* escritos en un *lenguaje de programación* para poder *ejecutarlos* en un *computador*.

Seis términos técnicos en el primer párrafo. No está mal. Vayamos paso a paso: empezaremos por presentar en qué consiste, básicamente, un computador.

1.1. Computadores

El diccionario de la Real Academia define computador electrónico como «Máquina electrónica, analógica o digital, dotada de una memoria de gran capacidad y de métodos de tratamiento de la información, capaz de resolver problemas matemáticos y lógicos mediante la utilización automática de programas informáticos».

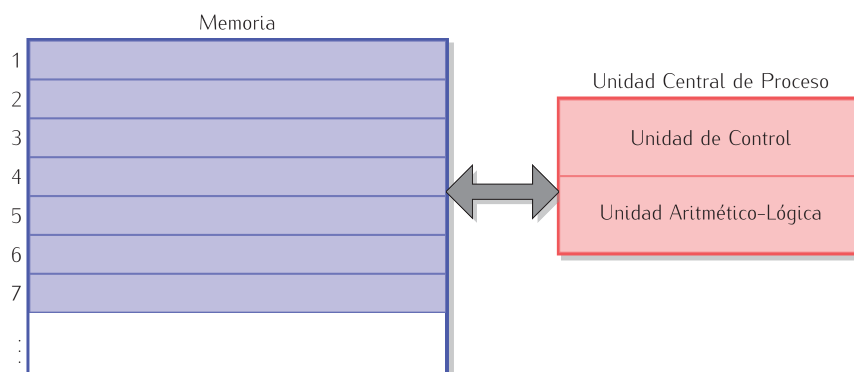
La propia definición nos da indicaciones acerca de algunos elementos básicos del computador:

- la memoria,

- y algún dispositivo capaz de efectuar cálculos matemáticos y lógicos.

La memoria es un gran almacén de información. En la memoria guardamos todo tipo de datos: valores numéricos, textos, imágenes, etc. El dispositivo encargado de efectuar operaciones matemáticas y lógicas, que recibe el nombre de *Unidad Aritmético-Lógica* (UAL), es como una calculadora capaz de trabajar con esos datos y producir, a partir de ellos, nuevos datos (el resultado de las operaciones). Otro dispositivo se encarga de transportar la información de la memoria a la UAL, de controlar a la UAL para que efectúe las operaciones pertinentes en el instante justo y de depositar los resultados en la memoria: la *Unidad de Control*. El conjunto que forman la Unidad de Control y la UAL se conoce por *Unidad Central de Proceso* (o CPU, del inglés «Central Processing Unit»).

Podemos imaginar la memoria como un armario enorme con cajones numerados y la CPU como una persona que, equipada con una calculadora (la UAL), es capaz de buscar operandos en la memoria, efectuar cálculos con ellos y dejar los resultados en la memoria.



Utilizaremos un lenguaje más técnico: cada uno de los «cajones» que conforman la memoria recibe el nombre de *celda* (de memoria) y el número que lo identifica es su *posición* o *dirección*, aunque a veces usaremos estos dos términos para referirnos también a la correspondiente celda.

Cada posición de memoria permite almacenar una secuencia de unos y ceros de tamaño fijo. ¿Por qué unos y ceros? Porque la tecnología actual de los computadores se basa en la sencillez con que es posible construir dispositivos binarios, es decir, que pueden adoptar dos posibles estados: encendido/apagado, hay corriente/no hay corriente, cierto/falso, uno/cero... ¿Es posible representar datos tan variados como números, textos, imágenes, etc. con solo unos y ceros? La respuesta es sí (aunque con ciertas limitaciones). Para entenderla mejor, es preciso que nos detengamos brevemente a considerar cómo se representa la información con valores binarios.

1.2. Codificación de la información

Una codificación asocia signos con los elementos de un conjunto a los que denominamos significados. En occidente, por ejemplo, codificamos los números de cero a nueve con el conjunto de signos $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Al hacerlo, ponemos en correspondencia estos símbolos con cantidades, es decir, con su significado: el símbolo «6» representa a la cantidad seis. El conjunto de signos no tiene por qué ser finito. Podemos combinar los dígitos en secuencias que ponemos en correspondencia con, por ejemplo, los números naturales. La sucesión de dígitos «99» forma un nuevo signo que asociamos a la cantidad noventa y nueve. Los ordenadores solo tienen dos signos básicos, $\{0, 1\}$, pero se pueden combinar en secuencias, así que no estamos limitados a solo dos posibles significados.

Una variable que solo puede tomar uno de los dos valores binarios recibe el nombre de *bit* (acrónimo del inglés «binary digit»). Es habitual trabajar con secuencias de bits de tamaño fijo. Una secuencia de 8 bits recibe el nombre de *byte* (aunque en español el término correcto es *octeto*, este no acaba de imponerse y se usa mucho más la voz inglesa). Con una secuencia de 8 bits podemos representar 256 (que es el valor de 2^8) significados diferentes. El rango $[0, 255]$ de valores naturales comprende 256 valores, así que podemos representar cualquiera de ellos con un patrón de 8 bits. Podríamos decidir, en principio, que la correspondencia entre bytes y valores naturales es completamente arbitraria. Así, podríamos decidir que la secuencia `00010011` representa, por ejemplo, el número natural 0 y que la secuencia `01010111` representa el valor 3. Aunque sea posible esta asociación arbitraria, no es deseable, pues complica enormemente efectuar operaciones con los valores. Sumar, por ejemplo, obligaría a tener memorizada una tabla que dijera cuál es el resultado de efectuar la operación con cada par de valores, ¡y hay 65.536 pares diferentes!

Los sistemas de representación posicional permiten establecer esa asociación entre secuencias de bits y valores numéricos naturales de forma sistemática. Centramos el discurso en secuencias de 8 bits, aunque todo lo que exponemos a continuación es válido para secuencias de otros tamaños[1]. El valor de una cadena de bits $b_7b_6b_5b_4b_3b_2b_1b_0$ es, en un sistema posicional convencional, $\sum_{i=0}^{7} b_i \cdot 2^i$. Así, la secuencia de bits `00001011` codifica el valor $0 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 8 + 2 + 1 = 11$. El bit de más a la

[1]Ocho bits ofrecen un rango de valores muy limitado. Es habitual en los ordenadores modernos trabajar con grupos de 4 bytes (32 bits) u 8 bytes (64 bits).

izquierda recibe el nombre de «bit más significativo» y el bit de más a la derecha se denomina «bit menos significativo».

▶ 1 ¿Cuál es el máximo valor que puede representarse con 16 bits y un sistema de representación posicional como el descrito? ¿Qué secuencia de bits le corresponde?

▶ 2 ¿Cuántos bits se necesitan para representar los números del 0 al 18, ambos inclusive?

El sistema posicional es especialmente adecuado para efectuar ciertas operaciones aritméticas. Tomemos por caso la suma. Hay una «tabla de sumar» en binario que te mostramos a continuación:

sumandos		suma	acarreo
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

El acarreo no nulo indica que un dígito no es suficiente para expresar la suma de dos bits y que debe añadirse el valor uno al bit que ocupa una posición más a la izquierda. Para ilustrar la sencillez de la adición en el sistema posicional, hagamos una suma de dos números de 8 bits usando esta tabla. En este ejemplo sumamos los valores 11 y 3 en su representación binaria:

```
  00001011
+ 00000011
  --------
```

Empezamos por los bits menos significativos. Según la tabla, la suma 1 y 1 da 0 con acarreo 1:

```
Acarreo        1
     00001011
   + 00000011
     --------
            0
```

El segundo dígito empezando por la derecha toma el valor que resulta de sumar a 1 y 1 el acarreo que arrastramos. O sea, 1 y 1 es 0 con acarreo 1, pero al sumar el acarreo que arrastramos de la anterior suma de bits, el resultado final es 1 con acarreo 1:

```
Acarreo       1 1
     00001011
   + 00000011
     --------
           10
```

Ya te habrás hecho una idea de la sencillez del método. De hecho, ya lo conoces bien, pues el sistema de numeración que aprendiste en la escuela es también posicional, solo que usando diez dígitos diferentes en lugar de dos, así que el procedimiento de suma es esencialmente idéntico. He aquí el resultado final, que es la secuencia de bits 00001110, o sea, el valor 14:

```
Acarreo       1 1
     00001011
   + 00000011
     --------
     00001110
```

La circuitería electrónica necesaria para implementar un sumador que actúe como el descrito es extremadamente sencilla.

▶ 3 Calcula las siguientes sumas de números codificados con 8 bits en el sistema posicional:

a) 01111111 + 00000001

b) `01010101 + 10101010`

c) `00000011 + 00000001`

Debes tener en cuenta que la suma de dos números de 8 bits puede proporcionar una cantidad que requiere 9 bits. Suma, por ejemplo, las cantidades 255 (en binario de 8 bits es 11111111) y 1 (que en binario es 00000001):

```
Acarreo  1 1 1 1 1 1 1
          11111111
      +   00000001
        (1)00000000
```

El resultado es la cantidad 256, que en binario se expresa con 9 bits, no con 8. Decimos en este caso que la suma ha producido un *desbordamiento*. Esta anomalía debe ser tenida en cuenta cuando se usa o programa un ordenador.

Hasta el momento hemos visto cómo codificar valores positivos. ¿Podemos representar también cantidades negativas? La respuesta es sí. Consideremos brevemente tres formas de hacerlo. La primera es muy intuitiva: consiste en utilizar el bit más significativo para codificar el signo; si vale 0, por ejemplo, el número expresado con los restantes bits es positivo (con la representación posicional que ya conoces), y si vale 1, es negativo. Por ejemplo, el valor de 00000010 es 2 y el de 10000010 es −2. Efectuar sumas con valores positivos y negativos resulta relativamente complicado si codificamos así el signo de un número. Esta mayor complicación se traslada también a la circuitería necesaria. Mala cosa.

Otra forma de codificar cantidades positivas y negativas es el denominado «complemento a uno». Consiste en lo siguiente: se toma la representación posicional de un número (que debe poder expresarse con 7 bits) y se invierten todos sus bits si es negativo. La suma de números codificados así es relativamente sencilla: se efectúa la suma convencional y, si no se ha producido un desbordamiento, el resultado es el valor que se deseaba calcular; pero si se produce un desbordamiento, la solución se obtiene sumando el valor 1 al resultado de la suma (sin tener en cuenta ya el bit desbordado). Veámoslo con un ejemplo. Sumemos el valor 3 al valor −2 en complemento a uno:

```
Acarreo  1 1 1 1 1 1 1
          00000011
      +   11111101
        (1)00000000
               ↓
          00000000
      +   00000001
          00000001
```

La primera suma ha producido un desbordamiento. El resultado correcto resulta de sumar una unidad a los 8 primeros bits.

La codificación en complemento a uno tiene algunas desventajas. Una de ellas es que hay dos formas de codificar el valor 0 (con 8 bits, por ejemplo, tanto 00000000 como 11111111 representan el valor 0) y, por tanto, solo podemos representar 255 valores ($[-127, 127]$), en lugar de 256. Pero el principal inconveniente es la potencial lentitud con que se realizan operaciones como la suma: cuando se produce un desbordamiento se han de efectuar dos adiciones, es decir, se ha de invertir el doble de tiempo para completar la operación.

Una codificación alternativa (y que es la utilizada en los ordenadores) es la denominada «complemento a dos». Para cambiar el signo a un número hemos de invertir todos sus bits *y sumar* 1 *al resultado*. Esta codificación, que parece poco natural, tiene las ventajas de que solo hay una forma de representar el valor nulo (el rango de valores representados es $[-128, 127]$) y de que una sola operación de suma basta siempre para obtener el resultado correcto de una adición. Repitamos el ejemplo anterior. Sumemos 3 y −2, pero en complemento a dos:

```
Acarreo   1 1 1 1 1 1
           00000011
   +       11111110
        (1)00000001
```

Si ignoramos el bit desbordado, el resultado es correcto.

▶ 4 Codifica en complemento a dos de 8 bits los siguientes valores:

a) 4

b) −4

c) 0

d) 127

e) 1

f) −1

▶ 5 Efectúa las siguientes sumas y restas en complemento a dos de 8 bits:

a) $4+4$

b) $-4+3$

c) $127-128$

d) $127-127$

e) $1-1$

f) $1-2$

Bueno, ya hemos hablado bastante acerca de cómo codificar números (aunque más adelante ofreceremos alguna reflexión acerca de cómo representar valores con parte fraccional). Preocupémonos por un instante acerca de cómo representar texto. Hay una tabla que pone en correspondencia 127 símbolos con secuencias de bits y que se ha asumido como estándar. Entre esos símbolos se encuentran todas las letras del alfabeto inglés, tanto en minúscula como en mayúscula, signos de puntuación y otros que puedes encontrar en el teclado. Es la denominada tabla ASCII, cuyo nombre corresponde a las siglas de «American Standard Code for Information Interchange». La correspondencia entre secuencias de bits y caracteres determinada por la tabla es arbitraria, pero aceptada como estándar. La letra «a», por ejemplo, se codifica con la secuencia de bits `01100001` y la letra «A» se codifica con `01000001`.

El texto se puede codificar, pues, como una secuencia de bits. Aquí tienes el texto «`Hola`» codificado con la tabla ASCII:

```
01001000 01101111 01101100 01100001
```

Cuando aparece ese texto en pantalla, no vemos esas secuencias de bits: no entenderíamos nada. En pantalla vemos una sucesión de gráficos, uno que corresponde a la letra «H», seguido de otro que corresponde a la letra «o»… Estos gráficos son patrones de píxeles almacenados en la memoria del ordenador y que se muestran en la pantalla. Un bit de valor 0 puede mostrarse como color blanco y un bit de valor 1 como color negro. La letra «H» que ves en pantalla, por ejemplo, podría no ser más que la visualización de este patrón de bits:

```
01000010
01000010
01000010
01111110
01000010
01000010
01000010
```

La tabla ASCII al completo

Estos son los valores (en base 10) de cada símbolo en la tabla ASCII:

0 nul	1 soh	2 stx	3 etx	4 eot	5 enq	6 ack	7 bel
8 bs	9 ht	10 nl	11 vt	12 np	13 cr	14 so	15 si
16 dle	17 dc1	18 dc2	19 dc3	20 dc4	21 nak	22 syn	23 etb
24 can	25 em	26 sub	27 esc	28 fs	29 gs	30 rs	31 us
32 sp	33 !	34 "	35 #	36 $	37 %	38 &	39 '
40 (	41)	42 *	43 +	44 ,	45 -	46 .	47 /
48 0	49 1	50 2	51 3	52 4	53 5	54 6	55 7
56 8	57 9	58 :	59 ;	60 <	61 =	62 >	63 ?
64 @	65 A	66 B	67 C	68 D	69 E	70 F	71 G
72 H	73 I	74 J	75 K	76 L	77 M	78 N	79 O
80 P	81 Q	82 R	83 S	84 T	85 U	86 V	87 W
88 X	89 Y	90 Z	91 [	92 \	93]	94 ^	95 _
96 `	97 a	98 b	99 c	100 d	101 e	102 f	103 g
104 h	105 i	106 j	107 k	108 l	109 m	110 n	111 o
112 p	113 q	114 r	115 s	116 t	117 u	118 v	119 w
120 x	121 y	122 z	123 {	124 |	125 }	126 ~	127 del

Los 32 primeros elementos de la tabla corresponden a los denominados «caracteres de control» y se diseñaron para controlar los dispositivos en los que se muestra el texto. La mayor parte de ellos son obsoletos, pues se concibieron cuando lo normal era que el texto producido por un ordenador se mostrara en un teletipo, una impresora, tarjetas perforadas o una consola. El carácter `bel`, por ejemplo, hacía sonar la campanita del teletipo (`bel` es abreviatura de «bell»). En un ordenador de hoy, ese carácter puede disparar un pitido. El carácter con número decimal 32 (en binario 00100000) es el espacio en blanco (`sp` es abreviatura de «space»). Del 33 al 47 tenemos un juego de caracteres variados (signos de puntuación, el signo del dólar, la almohadilla, algunos operadores matemáticos, etc.). Del 48 al 57 encontramos los dígitos del sistema decimal. Del 58 al 64 hay otros caracteres especiales (signos de puntuación, de comparación y la arroba). Empieza entonces el alfabeto en mayúsculas, que va del código 65 al 90). Del 91 al 96 volvemos a encontrar símbolos variados y el alfabeto en minúsculas sigue a estos con los símbolos del 97 al 122. La tabla se completa con otros cuatro símbolos variados y un último carácter de control: el que representa la acción de borrado de un carácter (`del` es abreviatura de «delete»).

En la memoria del ordenador se dispone de un patrón de bits para cada carácter[2]. Cuando se detecta el código ASCII 01001000, se muestra en pantalla el patrón de bits correspondiente a la representación gráfica de la «H». Truculento, pero eficaz.

No solo podemos representar caracteres con patrones de píxeles: todos los gráficos de ordenador son simples patrones de píxeles dispuestos como una matriz.

Como puedes ver, sí basta con ceros y unos para codificar la información que manejamos en un ordenador: números, texto, imágenes, etc.

1.3. Programas y lenguajes de programación

Antes de detenernos a hablar de la codificación de la información estábamos comentando que la memoria es un gran almacén con cajones numerados, es decir, identificables con valores numéricos: sus respectivas direcciones. En cada cajón se almacena una secuencia de bits de tamaño fijo. La CPU, el «cerebro» del ordenador, es capaz de ejecutar acciones especificadas mediante secuencias de *instrucciones*. Una instrucción describe una acción muy simple, del estilo de «suma esto con aquello», «multiplica las cantidades que hay en tal y cual posición de memoria», «deja el resultado en tal dirección de memoria», «haz una copia del dato de esta dirección en esta otra dirección», «averigua si la cantidad almacenada en determinada dirección es negativa», etc. Las instrucciones se representan mediante combinaciones particulares de unos y ceros (valores binarios) y, por tanto, se pueden almacenar en la memoria.

Combinando inteligentemente las instrucciones en una secuencia podemos hacer que la CPU ejecute cálculos más complejos. Una secuencia de instrucciones es un *programa*. Si hay una

[2]La realidad es cada vez más compleja. Los sistemas modernos almacenan los caracteres en memoria de otra forma, pero hablar de ello supone desviarnos mucho de lo que queremos contar.

instrucción para multiplicar pero ninguna para elevar un número al cubo, podemos construir un programa que efectúe este último cálculo a partir de las instrucciones disponibles. He aquí, grosso modo, una secuencia de instrucciones que calcula el cubo a partir de productos:

1) Toma el número y multiplícalo por sí mismo.

2) Multiplica el resultado de la última operación por el número original.

Las secuencias de instrucciones que el ordenador puede ejecutar reciben el nombre de *programas en código de máquina*, porque el *lenguaje de programación* en el que están expresadas recibe el nombre de *código de máquina*. Un lenguaje de programación es cualquier sistema de notación que permite expresar programas.

1.3.1. Código de máquina

El código de máquina codifica las secuencias de instrucciones como sucesiones de unos y ceros que siguen ciertas reglas. Cada familia de ordenadores dispone de su propio repertorio de instrucciones, es decir, de su propio código de máquina.

Un programa que, por ejemplo, calcula la media de tres números almacenados en las posiciones de memoria 10, 11 y 12, respectivamente, y deja el resultado en la posición de memoria 13, podría tener el siguiente aspecto expresado de forma comprensible para nosotros:

	Memoria
1	Sumar contenido de direcciones 10 y 11 y dejar resultado en dirección 13
2	Sumar contenido de direcciones 13 y 12 y dejar resultado en dirección 13
3	Dividir contenido de dirección 13 por 3 y dejar resultado en dirección 13
4	Detener
5	
6	
7	
⋮	

En realidad, el contenido de cada dirección estaría codificado como una serie de unos y ceros, así que el aspecto real de un programa como el descrito arriba podría ser este:

	Memoria
1	10101011 00001010 00001011 00001101
2	10101011 00001101 00001100 00001101
3	00001110 00001101 00000011 00001101
4	00000000 00000000 00000000 00000000
5	
6	
7	
⋮	

La CPU es un ingenioso sistema de circuitos electrónicos capaz de interpretar el significado de cada una de esas secuencias de bits y llevar a cabo las acciones que codifican. Cuando la CPU ejecuta el programa empieza por la instrucción contenida en la primera de sus posiciones de memoria. Una vez ha ejecutado una instrucción, pasa a la siguiente, y sigue así hasta encontrar una instrucción que detenga la ejecución del programa.

Supongamos que en las direcciones de memoria 10, 11 y 12 se han almacenado los valores 5, 10 y 6, respectivamente. Representamos así la memoria:

Memoria

1	`Sumar contenido de direcciones 10 y 11 y dejar resultado en dirección 13`
2	`Sumar contenido de direcciones 13 y 12 y dejar resultado en dirección 13`
3	`Dividir contenido de dirección 13 por 3 y dejar resultado en dirección 13`
4	`Detener`
⋮	
10	5
11	10
12	6
⋮	

Naturalmente, los valores de las posiciones 10, 11 y 12 estarán codificados en binario, aunque hemos optado por representarlos en base 10 en aras de una mayor claridad.

La ejecución del programa procede del siguiente modo. En primer lugar, se ejecuta la instrucción de la dirección 1, que dice que tomemos el contenido de la dirección 10 (el valor 5), lo sumemos al de la dirección 11 (el valor 10) y dejemos el resultado (el valor 15) en la dirección de memoria 13. Tras ejecutar esta primera instrucción, la memoria queda así:

Memoria

1	`Sumar contenido de direcciones 10 y 11 y dejar resultado en dirección 13`
2	`Sumar contenido de direcciones 13 y 12 y dejar resultado en dirección 13`
3	`Dividir contenido de dirección 13 por 3 y dejar resultado en dirección 13`
4	`Detener`
⋮	
10	5
11	10
12	6
13	15
⋮	

A continuación, se ejecuta la instrucción de la dirección 2, que ordena que se tome el contenido de la dirección 13 (el valor 15), se sume al contenido de la dirección 12 (el valor 6) y se deposite el resultado (el valor 21) en la dirección 13. La memoria pasa a quedar en este estado.

Memoria

1	`Sumar contenido de direcciones 10 y 11 y dejar resultado en dirección 13`
2	`Sumar contenido de direcciones 13 y 12 y dejar resultado en dirección 13`
3	`Dividir contenido de dirección 13 por 3 y dejar resultado en dirección 13`
4	`Detener`
⋮	
10	5
11	10
12	6
13	21
⋮	

Ahora, la tercera instrucción dice que hemos de tomar el valor de la dirección 13 (el valor 21), dividirlo por 3 y depositar el resultado (el valor 7) en la dirección 13. Este es el estado en que queda la memoria tras ejecutar la tercera instrucción:

Memoria

1	`Sumar contenido de direcciones 10 y 11 y dejar resultado en dirección 13`
2	`Sumar contenido de direcciones 13 y 12 y dejar resultado en dirección 13`
3	`Dividir contenido de dirección 13 por 3 y dejar resultado en dirección 13`
4	`Detener`
⋮	
10	5
11	10
12	6
13	7
⋮	

Y finalmente, la CPU detiene la ejecución del programa, pues se encuentra con la instrucción `Detener` en la dirección 4.

▶ 6 Ejecuta paso a paso el mismo programa con los valores 2, −2 y 0 en las posiciones de memoria 10, 11 y 12, respectivamente.

▶ 7 Diseña un programa que calcule la media de cinco números depositados en las posiciones de memoria que van de la 10 a la 14 y que deje el resultado en la dirección de memoria 15. Recuerda que la media $\bar{x}$ de cinco números x_1, x_2, x_3, x_4 y x_5 es

$$\bar{x} = \frac{\sum_{i=1}^{5} x_i}{5} = \frac{x_1 + x_2 + x_3 + x_4 + x_5}{5}.$$

▶ 8 Diseña un programa que calcule la varianza de cinco números depositados en las posiciones de memoria que van de la 10 a la 14 y que deje el resultado en la dirección de memoria 15. La varianza, que se denota con σ^2, es

$$\sigma^2 = \frac{\sum_{i=1}^{5} (x_i - \bar{x})^2}{5},$$

donde $\bar{x}$ es la media de los cinco valores. Supón que existe una instrucción «Multiplicar el contenido de dirección a por el contenido de dirección b y dejar el resultado en dirección c».

¿Qué instrucciones podemos usar para confeccionar programas? Ya hemos dicho que el ordenador solo sabe ejecutar instrucciones muy sencillas. En nuestro ejemplo, solo hemos utilizado tres instrucciones distintas:

- una instrucción de suma de la forma «`Sumar contenido de direcciones` p `y` q `y dejar resultado en dirección` r»;
- una instrucción de división de la forma «`Dividir contenido de dirección` p `por` q `y dejar resultado en dirección` r»;
- y una instrucción que indica que se ha llegado al final del programa: `Detener`.

¡Pocos programas interesantes podemos hacer con tan solo estas tres instrucciones! Naturalmente, en un código de máquina hay instrucciones que permiten efectuar sumas, restas, divisiones y otras muchas operaciones. Y hay, además, instrucciones que permiten escoger qué instrucción se ejecutará a continuación, bien directamente, bien en función de si se cumple o no determinada condición (por ejemplo, «`Si el último resultado es negativo, pasar a ejecutar la instrucción de la posición` p»).

1.3.2. Lenguaje ensamblador

En los primeros tiempos de la informática los programas se introducían en el ordenador directamente en código de máquina, indicando uno por uno el valor de los bits de cada una de las posiciones de memoria. Para ello se insertaban manualmente cables en un panel de conectores: cada cable insertado en un conector representaba un uno y cada conector sin cable representaba un cero. Como puedes imaginar, programar así un computador resultaba una tarea ardua, extremadamente tediosa y propensa a la comisión de errores. El más mínimo fallo conducía a un programa incorrecto. Pronto se diseñaron notaciones que simplificaban la programación: cada instrucción de código de máquina se representaba mediante un *código mnemotécnico*, es decir, una abreviatura fácilmente identificable con el propósito de la instrucción.

Por ejemplo, el programa desarrollado antes se podría representar como el siguiente texto:

```
SUM #10, #11, #13
SUM #13, #12, #13
DIV #13, 3, #13
FIN
```

En este lenguaje la palabra `SUM` representa la instrucción de sumar, `DIV` la de dividir y `FIN` representa la instrucción que indica que debe finalizar la ejecución del programa. La almohadilla (`#`) delante de un número indica que deseamos acceder al contenido de la posición de memoria cuya dirección es dicho número. Los caracteres que representan el programa se introducen en la memoria del ordenador con la ayuda de un teclado y cada letra se almacena en una posición de memoria como una combinación particular de unos y ceros (su código ASCII, por ejemplo).

Pero ¿cómo se puede ejecutar ese tipo de programa si la secuencia de unos y ceros que la describe como texto no constituye un programa válido en código de máquina? Con la ayuda de otro programa: el *ensamblador*. El ensamblador es un programa traductor que lee el contenido de las direcciones de memoria en las que hemos almacenado códigos mnemotécnicos y escribe en otras posiciones de memoria sus instrucciones asociadas en código de máquina.

El repertorio de códigos mnemotécnicos traducible a código de máquina y las reglas que permiten combinarlos, expresar direcciones, codificar valores numéricos, etc., recibe el nombre de *lenguaje ensamblador*, y es otro lenguaje de programación.

1.3.3. ¿Un programa diferente para cada ordenador?

Cada CPU tiene su propio juego de instrucciones y, en consecuencia, un código de máquina y uno o más lenguajes ensambladores propios. Un programa escrito para una CPU de la marca Intel no funcionará en una CPU diseñada por otro fabricante, como Motorola[3]. ¡Incluso diferentes versiones de una misma CPU tienen juegos de instrucciones que no son totalmente compatibles entre sí!: los modelos más evolucionados de una familia de CPU pueden incorporar instrucciones que no se encuentran en los más antiguos[4].

Si queremos que un programa se ejecute en más de un tipo de ordenador, ¿habrá que escribirlo de nuevo para cada CPU particular? Durante mucho tiempo se intentó definir algún tipo de «lenguaje ensamblador universal», es decir, un lenguaje cuyos códigos mnemotécnicos, sin corresponderse con los del código de máquina de ningún ordenador concreto, fuesen fácilmente traducibles al código de máquina de cualquier ordenador. Disponer de dicho lenguaje permitiría escribir los programas una sola vez y ejecutarlos en diferentes ordenadores tras efectuar las correspondientes traducciones a cada código de máquina con diferentes programas ensambladores.

Si bien la idea es en principio interesante, presenta serios inconvenientes:

- Un lenguaje ensamblador universal no puede tener en cuenta cómo se diseñarán ordenadores en un futuro y qué tipo de instrucciones soportarán, así que posiblemente quede obsoleto en poco tiempo.

[3] A menos que la CPU se haya diseñado expresamente para reproducir el funcionamiento de la primera, como ocurre con los procesadores de AMD, diseñados con el objetivo de ejecutar el código de máquina de los procesadores de Intel.

[4] Por ejemplo, añadiendo instrucciones que faciliten la programación de aplicaciones multimedia (como ocurre con los Intel Pentium MMX y modelos posteriores) prácticamente impensables cuando se diseñó la primera CPU de la familia (el Intel 8086).

¡Hola, mundo!

Nos gustaría mostrarte el aspecto de los programas escritos en lenguajes ensambladores reales con un par de ejemplos. Es una tradición ilustrar los diferentes lenguajes de programación con un programa sencillo que se limita a mostrar por pantalla el mensaje «Hello, World!» («¡Hola, mundo!»), así que la seguiremos. He aquí ese programa escrito en los lenguajes ensambladores de dos CPU distintas: a mano izquierda, el de los procesadores 80x86 de Intel (cuyos últimos representantes por el momento son la familia de procesadores i5 e i7) y, a mano derecha, el de los procesadores de la familia Motorola 68000 (que es el procesador de los primeros ordenadores Apple Macintosh).

```
    .data
msg:
    .string "Hello, World!\n"
len:
    .long . - msg
.text
.globl _start
_start:
    push $len
    push $msg
    push $1
    movl $0x4, %eax
    call _syscall
    addl $12, %esp
    push $0
    movl $0x1, %eax
    call _syscall
_syscall:
    int  $0x80
    ret
```

```
start:
  move.l #msg,-(a7)
  move.w #9,-(a7)
  trap #1
  addq.l #6,a7
  move.w #1,-(a7)
  trap #1
  addq.l  #2,a7
  clr -(a7)
  trap #1
  msg: dc.b "Hello, World!",10,13,0
```

Como puedes ver, ambos programas presentan un aspecto muy diferente. Por otra parte, los dos son bastante largos (entre 10 y 20 líneas) y de difícil comprensión, a menos que se cuente con conocimiento preciso de lo que hace cada instrucción y las reglas sintácticas de cada lenguaje ensamblador.

- Programar en lenguaje ensamblador (incluso en ese supuesto lenguaje ensamblador universal) es complicadísimo por los numerosos detalles que deben tenerse en cuenta.

Además, puestos a diseñar un lenguaje de programación general, ¿por qué no utilizar un lenguaje natural, es decir un lenguaje como el castellano o el inglés? Programar un computador consistiría, simplemente, en escribir (¡o pronunciar frente a un micrófono!) un texto en el que indicásemos qué deseamos que haga el ordenador usando el mismo lenguaje con que nos comunicamos con otras personas. Un programa informático podría encargarse de traducir nuestras frases al código de máquina, del mismo modo que un programa ensamblador traduce lenguaje ensamblador a código de máquina. Es una idea atractiva, pero que queda lejos de lo que sabemos hacer por varias razones:

- La complejidad intrínseca de las construcciones de los lenguajes naturales dificulta enormemente el *análisis sintáctico* de las frases, es decir, comprender su estructura y cómo se relacionan entre sí los diferentes elementos que las constituyen.
- El *análisis semántico*, es decir, la comprensión del significado de las frases, es aún más complicado. Las ambigüedades e imprecisiones del lenguaje natural hacen que sus frases presenten, fácilmente, diversos significados, aun cuando las podamos analizar sintácticamente. (¿Cuántos significados tiene la frase «Trabaja en un banco.»?) Sin una buena comprensión del significado no es posible efectuar una traducción aceptable.

1.3.4. Lenguajes de programación de alto nivel

Hay una solución intermedia: podemos diseñar lenguajes de programación que, sin ser tan potentes y expresivos como los lenguajes naturales, eliminen buena parte de la complejidad propia de los lenguajes ensambladores y estén bien adaptados al tipo de problemas que podemos

resolver con los computadores: los denominados *lenguajes de programación de alto nivel*. El calificativo «de alto nivel» señala su independencia de un ordenador concreto. Por contraposición, los códigos de máquina y los lenguajes ensambladores se denominan *lenguajes de programación de bajo nivel*.

He aquí el programa que calcula la media de tres números en un lenguaje de alto nivel típico (Python):

```
a = 5
b = 10
c = 6
media = (a + b + c) / 3
```

Las tres primeras líneas definen los tres valores y la cuarta calcula la media. Como puedes ver, resulta mucho más legible que un programa en código de máquina o en un lenguaje ensamblador.

Para cada lenguaje de alto nivel y para cada CPU se puede escribir un programa que se encargue de traducir las instrucciones del lenguaje de alto nivel a instrucciones de código de máquina, con lo que se consigue la deseada independencia de los programas con respecto a los diferentes sistemas computadores. Solo habrá que escribir una versión del programa en un lenguaje de programación de alto nivel y la traducción de ese programa al código de máquina de cada CPU se realizará automáticamente.

1.3.5. Compiladores e intérpretes

Hemos dicho que los lenguajes de alto nivel se traducen automáticamente a código de máquina, sí, pero has de saber que hay dos tipos diferentes de traductores dependiendo de su modo de funcionamiento: *compiladores* e *intérpretes*.

Un *compilador* lee completamente un programa en un lenguaje de alto nivel y lo traduce en su integridad a un programa de código de máquina equivalente. El programa de código de máquina resultante se puede ejecutar cuantas veces se desee, sin necesidad de volver a traducir el programa original.

Un *intérprete* actúa de un modo distinto: lee un programa escrito en un lenguaje de alto nivel instrucción a instrucción y, para cada una de ellas, efectúa una traducción a las instrucciones de código de máquina equivalentes y las ejecuta inmediatamente. No hay un proceso de traducción separado por completo del de ejecución. Cada vez que ejecutamos el programa con un intérprete, se repite el proceso de traducción y ejecución, ya que ambos son simultáneos.

Compiladores e intérpretes... de idiomas

Puede resultarte de ayuda establecer una analogía entre compiladores e intérpretes de lenguajes de programación y traductores e intérpretes de idiomas.

Un compilador actúa como un traductor que recibe un libro escrito en un idioma determinado (lenguaje de alto nivel) y escribe un nuevo libro que, con la mayor fidelidad posible, contiene una traducción del texto original a otro idioma (código de máquina). El proceso de traducción (compilación) tiene lugar una sola vez y podemos leer el libro (ejecutar el programa) en el idioma destino (código de máquina) cuantas veces queramos.

Un intérprete de programas actúa como su homónimo en el caso de los idiomas. Supón que se imparte una conferencia en inglés en diferentes ciudades y un intérprete ofrece su traducción simultánea al castellano. Cada vez que la conferencia es pronunciada, el intérprete debe realizar nuevamente la traducción. Es más, la traducción se produce sobre la marcha, frase a frase, y no de un tirón al final de la conferencia. Del mismo modo actúa un intérprete de un lenguaje de programación: traduce cada vez que ejecutamos el programa y además lo hace instrucción a instrucción.

Por regla general, los intérpretes ejecutarán los programas más lentamente, pues al tiempo de ejecución del código de máquina se suma el que consume la traducción simultánea. Además, un compilador puede examinar el programa de alto nivel abarcando más de una instrucción cada vez, por lo que es capaz de producir mejores traducciones. Un programa interpretado suele ser mucho más lento que otro equivalente que haya sido compilado (¡típicamente entre 2 y 100 veces más lento!).

Si tan lento resulta interpretar un programa, ¿por qué no se usan únicamente compiladores? Es pronto para que entiendas las razones, pero, por regla general, los intérpretes permiten una mayor flexibilidad que los compiladores y ciertos lenguajes de programación de alto nivel han sido diseñados para explotar esa mayor flexibilidad. Otros lenguajes de programación, por contra, sacrifican la flexibilidad en aras de una mayor velocidad de ejecución. Aunque nada impide que compilemos o interpretemos cualquier lenguaje de programación, ciertos lenguajes se consideran apropiados para que la traducción se lleve a cabo con un compilador y otros no. Es más apropiado hablar, pues, de lenguajes de programación *típicamente* interpretados y lenguajes de programación *típicamente* compilados. Entre los primeros podemos citar Python, BASIC, Perl, Tcl, Ruby, Bash, Java o Lisp. Entre los segundos, C, C#, Pascal, C++ o Fortran.[5]

En el primer curso de los grados en Ingeniería Informática y en Matemática Computacional de la Universitat Jaume I aprenderemos a programar usando dos lenguajes de programación distintos: uno interpretado, Python, y otro, Java, que puede ser interpretado o compilado a partir de un lenguaje intermedio. Este volumen se dedica al estudio del lenguaje de programación Python.

1.3.6. Python

Existen muchos otros lenguajes de programación, ¿por qué aprender Python? Python presenta una serie de ventajas que lo hacen muy atractivo, tanto para su uso profesional como para el aprendizaje de la programación. Entre las más interesantes desde el punto de vista didáctico tenemos:

- Python es un lenguaje muy *expresivo*, es decir, los programas Python son muy compactos: un programa Python suele ser bastante más corto que su equivalente en lenguajes como C. (Python llega a ser considerado por muchos un *lenguaje de programación de muy alto nivel*).

- Python es muy *legible*. La sintaxis de Python es muy elegante y permite la escritura de programas cuya lectura resulta más fácil que si utilizáramos otros lenguajes de programación.

- Python ofrece un *entorno interactivo* que facilita la realización de pruebas y ayuda a despejar dudas acerca de ciertas características del lenguaje.

- El *entorno de ejecución* de Python *detecta muchos de los errores* de programación que escapan al control de los compiladores y proporciona información muy rica para detectarlos y corregirlos.

- Python puede usarse como lenguaje *imperativo procedimental* o como lenguaje *orientado a objetos*.

- Posee un *rico juego de estructuras de datos* que se pueden manipular de modo sencillo.

Estas características hacen que sea relativamente fácil traducir métodos de cálculo a programas Python.

Los lenguajes de programación no permanecen inmutables a lo largo del tiempo: evolucionan. Python no es una excepción. A partir de la experiencia con una versión del lenguaje y de la influencia que ejercen otros lenguajes sobre los programadores, hay una presión constante por hacer que el lenguaje ofrezca nuevas capacidades o simplifique el modo en el que se expresan ciertos cálculos. Python fue diseñado inicialmente por Guido van Rossum a partir de su experiencia colaborando con otros en el desarrollo de un lenguaje experimental: ABC. La World Wide Web aparecía al poco de crearse la primera versión de Python y ayudaba a poner en contacto a miles de programadores en todo el mundo. La elegancia de Python, unida a la aparición de un nuevo medio de comunicación entre especialistas, hicieron que un lenguaje que

[5] Lo cierto es que la mayoría de los lenguajes interpretados se traducen primero a un lenguaje intermedio que es el realmente interpretado. Ocurre, por ejemplo, con Python y Java. C# es aún más especial: los programas escritos en este lenguaje se traducen a un lenguaje intermedio que, a su vez, se traduce a código de máquina en cada ejecución, pero de una sola vez. Este modelo de compilación en dos etapas también ha pasado a ser corriente para Java.

no provenía de la academia o la industria tuviera un éxito inusitado. Hablamos de los años 90 del pasado siglo, década en la que fue tomando fuerza el concepto de «software libre».

Una activa comunidad de desarrolladores liderada por Guido van Rossum (quien sigue teniendo la última palabra en todas las decisiones) va mejorando el lenguaje progresivamente. Cada nueva versión se marca con una serie de números separados por puntos. Lee, si quieres, el cuadro titulado «Versiones» para entender más sobre la codificación tradicional de versiones de productos software.

Versiones

Los productos software evolucionan añadiendo funcionalidad o corrigiendo errores presentes en una versión determinada. Se ha adoptado un convenio para referirse a cada una de las versiones de un producto software: una serie de números separados por puntos. Típicamente se usan 2 números separados por un punto: el número principal («major version number») y el número secundario («minor version number»). La primera versión de un producto es la 1.0. Si se añade funcionalidad menor o se corrigen errores menores, el autor del software irá produciendo las versiones 1.1, 1.2, 1.3, etc. del producto (y que no acaba con la 1.9, pues puede seguir la 1.10, la 1.11, etc.). Se entiende que todas esas versiones son «compatibles hacia atrás», es decir, los datos que sirven para la versión 1.x son utilizables en toda versión 1.y donde $y > x$. En ocasiones se sacan versiones que contienen cambios tan pequeños que no merece avanzar ni siquiera el número secundario. Así, a la versión 1.6 le puede suceder la 1.6.1 y a esta la 1.6.2 para pasar luego a la 1.7. Cuando hay un cambio de funcionalidad importante, se inicia una nueva serie de versiones que comienza en la 2.0. A la 2.0 le sigue la 2.1 y así sucesivamente.

Pero no siempre las cosas son tan sencillas. El software que se produce en empresas suele considerar aspectos relacionados con el márketing y omitir o esconder la numeración que hemos presentado. El sistema operativo Microsoft Windows siguió el criterio que hemos descrito en un principio. Hubo un Microsoft Windows 1.0, un 1.1, un 2.0, un 3.0 y un 3.1. Incluso un 3.11. Pero la numeración se rompió aparentemente con Microsoft Windows 95. Decimos aparentemente porque internamente se mantuvo. Esa versión de Windows es la cuarta. La serie XP es la quinta. Microsoft Windows Vista es la versión 6.0. Y Microsoft Windows 7 es la… ¡6.1!

Hemos hablado únicamente de las versiones que se lanzan al público. Durante el desarrollo hay varias fases de pruebas antes de lanzar un producto. La primera versión que se hace pública es la versión «alfa». Así, un producto puede tener una versión «1.3 alfa» y hacerla pública para pedir a potenciales colaboradores que ayuden a detectar errores. No es raro que se programe lanzar dos versiones alfa. Por ejemplo «1.3 alfa 1» y «1.3 alfa 2», recogiendo la segunda correcciones a errores detectados en la primera o incorporando alguna mejora de última hora. Cuando el software empieza a considerarse robusto se pasa a las versiones beta, de las que también suele planificarse un par de versiones. En las versiones beta solo se corrigen errores y no se añade nueva funcionalidad. Y antes de salir al mercado, aún se programa una versión casi definitiva: la versión RC (por «Release Candidate» o «candidato para lanzamiento»). Nuestro hipotético producto conocería entonces una versión 1.3 RC. Si la RC contuviera aún numerosos fallos (lo que no sería buena señal), podría publicarse una segunda RC, es decir, una versión 1.3 RC2. No hay obligación de seguir este esquema de numeración, pero lo encontrarás con frecuencia en muchos productos.

En el momento de escribir estas líneas, se ha publicado la versión de Python 3.4, precedida por las versiones 3.4 alpha 1, 2, 3 y 4; 3.4 beta 1, 2 y 3; 3.4 RC 1, 2 y 3.

La primera versión de Python con un uso extendido es la 1.5. Una gran comunidad de desarrolladores, liderada por el autor original, trabaja continuamente en la mejora del lenguaje. Aproximadamente cada año se hace pública una nueva versión de Python. ¡Tranquilo! No es que con cada versión cambie radicalmente el lenguaje de programación, sino que este se enriquece manteniendo en lo posible la compatibilidad con los programas escritos para versiones anteriores. Nosotros utilizaremos características de la versión 3.1 de Python, por lo que deberás utilizar esa versión o una superior.

Una ventaja fundamental de Python es la gratuidad de su intérprete. Puedes descargar el intérprete de la página web `http://www.python.org`. El intérprete de Python tiene versiones para prácticamente cualquier plataforma en uso: sistemas PC bajo Linux, sistemas PC bajo Microsoft Windows, sistemas Macintosh de Apple, etc.

Para que te vayas haciendo a la idea de qué aspecto presenta un programa completo en Python, te presentamos uno que calcula la media de tres números que introduce por teclado el

usuario y muestra el resultado por pantalla:

```
a = float(input('Dame un número:'))
b = float(input('Dame otro número:'))
c = float(input('Y ahora, uno más:'))
media = (a + b + c) / 3
print('La media es', media)
```

Python 2.x y Python 3.x

Estamos en un momento especial en el desarrollo de Python: conviven dos «ramas» del lenguaje que evolucionan simultáneamente. Una rama, la que se conoce como 2.x, publicó hace varios meses la versión 2.7.6 del lenguaje. La otra, conocida como 3.x, acaba de publicar la versión 3.4.0. El lenguaje Python es básicamente el mismo en las dos ramas, pero hay algunas diferencias importantes que hacen que un programa escrito para la versión 2.7 no siempre funcione en la versión 3.4. No ocurre lo mismo con los programas escritos para versiones distintas de la misma serie, es decir, un programa escrito para la versión 2.4, por ejemplo, debería funcionar perfectamente con un intérprete de cualquier versión posterior en la serie 2.x. Decimos que cada versión de la serie 2.x (o 3.x) presenta «compatibilidad hacia atrás» dentro de la misma serie.

¿Por qué hay una rama 3.x incompatible con programas escritos para Python 2.x? A lo largo de los años se fueron detectando fallos o aspectos poco elegantes del diseño de Python, cuya corrección o mejora supondría que los programas ya escritos dejaran de funcionar. Guido van Rossum hablaba entonces de una versión de Python ideal en la que todos los problemas estarían resueltos: Python 3000. El número 3000 era una referencia jocosa tanto a la versión en la que todo estaría resuelto como al año en el que se publicaría esa versión. Un buen día decidió no esperar tanto y anunció que Python crearía un desarrollo en dos ramas: la serie 2.x y la 3.x. Durante un tiempo, los programadores encontrarían mejoras en el lenguaje que no introducirían incompatibilidades (la serie 2.x), pero se animaba a que, en unos pocos años, todos fueran adaptando sus programas a la versión 3.x. Pasado un tiempo razonable, se cancelaría el desarrollo en la serie 2.x y solo evolucionaría la 3.x. Un plan sensato para no estancar el lenguaje y no fastidiar a todos los programadores que habían apostado por Python hace años.

En la última década Python ha experimentado un importantísimo aumento del número de programadores y empresas que lo utilizan. Google, por ejemplo, usa Python como uno de sus principales lenguajes de desarrollo (otros dos son Java y C++). Guido van Rossum, inventor de Python, trabaja en Google. También YouTube usa Python en sus sistemas de explotación. E Industrial Light & Magic (la empresa de efectos especiales de George Lucas) y Pixar recurren a Python en sus sistemas de producción cinematográfica. Y si alguna vez has usado BitTorrent, el popular sistema P2P, has de saber que está escrito en Python. La relación de empresas e instituciones que usa Python es inacabable: Intel, Hewlett-Packard, NASA, JPMorgan Chase, etc. Aquí tienes unas citas que encabezaron durante algún tiempo la web oficial de Python (`http://www.python.org`):

> Python ha sido parte importante de Google desde el principio, y lo sigue siendo a medida que el sistema crece y evoluciona. Hoy día, docenas de ingenieros de Google usan Python y seguimos buscando gente diestra en este lenguaje.
>
> Peter Norvig, director de calidad de búsquedas de Google Inc.

> Python juega un papel clave en nuestra cadena de producción. Sin él, un proyecto de la envergadura de «Star Wars: Episodio II» hubiera sido muy difícil de sacar adelante. Visualización de multitudes, proceso de lotes, composición de escenas... Python es lo que lo une todo.
>
> Tommy Brunette, director técnico senior de Industrial Light & Magic.

> Python está en todas partes de Industrial Light & Magic. Se usa para extender la capacidad de nuestras aplicaciones y para proporcionar la cola que las une. Cada imagen generada por computador que creamos incluye a Python en algún punto del proceso.
>
> Philip Peterson, ingeniero principal de I+D de Industrial Light & Magic.

1.3.7. Java

El lenguaje de programación Java es uno de los más utilizados en el mundo profesional. Se diseñó por técnicos de Sun Microsystems a mediados de los años 90 como un lenguaje y entorno de programación cuyos programas serían ejecutables en todo tipo de dispositivos: teléfonos móviles, televisores, microondas, etc. Al presentarse en público, Java permitía crear pequeños programas ejecutables en navegadores web: los denominados *applets*. En ese momento, la posibilidad de extender la experiencia de la navegación con aplicaciones interactivas era revolucionaria y dio un gran impulso al lenguaje. Pero finalmente Java se impuso en el mundo de las aplicaciones web del lado del servidor. Las aplicaciones web ofrecen servicios accesibles con el mismo protocolo de comunicación que usan los navegadores.

Finalmente no (solo) es el lenguaje en sí lo que hace productivo el trabajo de los desarrolladores: la colección de utilidades y bibliotecas que lo acompañan es fundamental. Java cuenta con un entorno de desarrollo muy rico y un gran conjunto de librerías de código, es decir, colecciones de funciones ya escritas que el programador usa para no tener que reinventar la rueda constantemente.

No obstante, programar con Java requiere un mayor esfuerzo que hacerlo con Python. Los programas Java son más extensos y requieren de una gran atención a muchos detalles por parte del programador.

La torre de Babel

Hemos dicho que los lenguajes de programación de alto nivel pretendían, entre otros objetivos, paliar el problema de que cada ordenador utilice su propio código de máquina. Puede que, en consecuencia, estés sorprendido por el número de lenguajes de programación citados. Pues los que hemos citado son unos pocos de los más utilizados: ¡hay centenares! ¿Por qué tantos?

El primer lenguaje de programación de alto nivel fue Fortran, que se diseñó en los primeros años 50 (y aún se utiliza hoy día, aunque en versiones evolucionadas). Fortran se diseñó con el propósito de traducir fórmulas matemáticas a código de máquina (de hecho, su nombre proviene de «FORmula TRANslator», es decir, «traductor de fórmulas»). Poco después se diseñaron otros lenguajes de programación con propósitos específicos: Cobol (Common Business Oriented Language), Lisp (List Processing language), etc. Cada uno de estos lenguajes hacía fácil la escritura de programas para solucionar problemas de ámbitos particulares: Cobol para problemas de gestión empresarial, Lisp para ciertos problemas de Inteligencia Artificial, etc. Hubo también esfuerzos para diseñar lenguajes de «propósito general», es decir, aplicables a cualquier dominio, como Algol 60 (Algorithmic Language). En la década de los 60 hicieron su aparición nuevos lenguajes de programación (Algol 68, Pascal, Simula 67, Snobol 4, etc.), pero quizá lo más notable de esta década fue que se sentaron las bases teóricas del diseño de compiladores e intérpretes. Cuando la tecnología para el diseño de estas herramientas se hizo accesible a más y más programadores hubo un auténtico estallido en el número de lenguajes de programación. Ya en 1969 se habían diseñado unos 120 lenguajes de programación y se habían implementado compiladores o intérpretes para cada uno de ellos.

La existencia de tantísimos lenguajes de programación creó una situación similar a la de la torre de Babel: cada laboratorio o departamento informático usaba un lenguaje de programación y no había forma de intercambiar programas.

Con los años se ha ido produciendo una selección de aquellos lenguajes de programación más adecuados para cada tipo de tarea y se han diseñado muchos otros que sintetizan lo aprendido de lenguajes anteriores. Los más utilizados hoy día son Java, C, C++, Python, Perl y PHP.

Si tienes curiosidad, puedes ver ejemplos del programa «Hello, world!» en más de 100 lenguajes de programación diferentes visitando la página `http://www.scriptol.com/programming/hello-world.php`

1.4. Más allá de los programas: algoritmos

Dos programas que resuelven el mismo problema expresados en el mismo o en diferentes lenguajes de programación pero que siguen, en lo fundamental, el mismo procedimiento, son dos *implementaciones* del mismo *algoritmo*. Un algoritmo es, sencillamente, una secuencia de pasos orientada a la consecución de un objetivo.

Cuando diseñamos un algoritmo podemos expresarlo en uno cualquiera de los numerosos lenguajes de programación de propósito general existentes. Sin embargo, ello resulta poco adecuado:

- no todos los programadores conocen todos los lenguajes y no hay consenso acerca de cuál es el más adecuado para expresar las soluciones a los diferentes problemas,

- cualquiera de los lenguajes de programación presenta particularidades que pueden interferir en una expresión clara y concisa de la solución a un problema.

Podemos expresar los algoritmos en lenguaje natural, pues el objetivo es comunicar un procedimiento resolutivo a otras personas y, eventualmente, traducirlos a algún lenguaje de programación. Si, por ejemplo, deseamos calcular la media de tres números leídos de teclado podemos seguir este algoritmo:

1) solicitar el valor del primer número,

2) solicitar el valor del segundo número,

3) solicitar el valor del tercer número,

4) sumar los tres números y dividir el resultado por 3,

5) mostrar el resultado.

Como puedes ver, esta secuencia de operaciones define exactamente el proceso que nos permite efectuar el cálculo propuesto y que ya hemos implementado como un programa en Python.

Los algoritmos son independientes del lenguaje de programación. Describen un procedimiento que puede ser implementado en cualquier lenguaje de programación de propósito general o, incluso, que puedes ejecutar a mano con lápiz, papel y, quizá, la ayuda de una calculadora.

¡Ojo! No es cierto que cualquier procedimiento descrito paso a paso pueda considerarse un algoritmo. Un algoritmo debe satisfacer ciertas condiciones. Una analogía con recetas de cocina (procedimientos para preparar platos) te ayudará a entender dichas restricciones.

Estudia esta primera receta:

1) poner aceite en una sartén,

2) encender el fuego,

3) calentar el aceite,

4) coger un huevo,

5) romper la cáscara,

6) verter el contenido del huevo en la sartén,

7) aderezar con sal,

8) esperar a que tenga buen aspecto.

En principio ya está: con la receta, sus ingredientes y los útiles necesarios somos capaces de cocinar un plato. Bueno, no del todo cierto, pues hay unas cuantas cuestiones que no quedan del todo claras en nuestra receta:

- ¿Qué tipo de huevo utilizamos?: ¿un huevo de gallina?, ¿un huevo de rana?

- ¿Cuánta sal utilizamos?: ¿una pizca?, ¿un kilo?

- ¿Cuánto aceite hemos de verter en la sartén?: ¿un centímetro cúbico?, ¿un litro?

- ¿Cuál es el resultado del proceso?, ¿la sartén con el huevo cocinado y el aceite?

En una receta de cocina hemos de dejar bien claro con qué ingredientes contamos y cuál es el resultado final. En un algoritmo hemos de precisar cuáles son los datos del problema (datos de entrada) y qué resultado vamos a producir (datos de salida).

Esta nueva receta corrige esos fallos:

- Ingredientes: 10 cc. de aceite de oliva, una gallina y una pizca de sal.
- Método:
  1) *esperar a que la gallina ponga un huevo*,
  2) poner aceite en una sartén,
  3) encender el fuego,
  4) calentar el aceite,
  5) coger el huevo,
  6) romper la cáscara,
  7) verter el contenido del huevo en la sartén,
  8) aderezar con sal,
  9) esperar a que tenga buen aspecto.
- Presentación: depositar el huevo frito, sin aceite, en un plato.

Pero la receta aún no está bien del todo. Hay ciertas indefiniciones en la receta:

1) ¿Cuán caliente ha de estar el aceite en el momento de verter el huevo?, ¿humeando?, ¿ardiendo?
2) ¿Cuánto hay que esperar?, ¿un segundo?, ¿hasta que el huevo esté ennegrecido?
3) Y aún peor, *¿estamos seguros de que la gallina pondrá un huevo?* Podría ocurrir que la gallina no pusiera huevo alguno.

Para que la receta esté completa, deberíamos especificar con *absoluta precisión* cada uno de los pasos que conducen a la realización del objetivo y, además, cada uno de ellos debería ser realizable en *tiempo finito*.

No basta con decir *más o menos* cómo alcanzar el objetivo: hay que decir *exactamente* cómo se debe ejecutar cada paso y, además, cada paso debe ser realizable en tiempo finito. Esta nueva receta corrige algunos de los problemas de la anterior, pero presenta otros de distinta naturaleza:

- Ingredientes: 10 cc. de aceite de oliva, un huevo de gallina y una pizca de sal.
- Método:
  1) poner aceite en una sartén,
  2) encender el fuego a medio gas,
  3) calentar el aceite hasta que humee ligeramente,
  4) coger un huevo,
  5) romper la cáscara *con el poder de la mente, sin tocar el huevo*,
  6) verter el contenido del huevo en la sartén,
  7) aderezar con sal,
  8) esperar a que tenga buen aspecto.
- Presentación: depositar el huevo frito, sin aceite, en un plato.

El quinto paso no es *factible*. Para romper un huevo has de utilizar algo más que «el poder de la mente». En todo algoritmo debes utilizar únicamente instrucciones que pueden llevarse a cabo.

He aquí una receta en la que todos los pasos son realizables:

- Ingredientes: 10 cc. de aceite de oliva, un huevo de gallina y una pizca de sal.
- Método:
  1) poner aceite en una sartén,
  2) *sintonizar una emisora musical en la radio,*
  3) encender el fuego a medio gas,
  4) *echar una partida al solitario,*
  5) calentar el aceite hasta que humee ligeramente,
  6) coger un huevo,
  7) romper la cáscara,
  8) verter el contenido del huevo en la sartén,
  9) aderezar con sal,
  10) esperar a que tenga buen aspecto.
- Presentación: depositar el huevo frito, sin aceite, en un plato.

En esta nueva receta hay acciones que, aunque expresadas con suficiente precisión y siendo realizables, no hacen nada *útil* para alcanzar nuestro objetivo (sintonizar la radio y jugar a las cartas). En un algoritmo, *cada paso dado debe conducir y acercarnos más a la consecución del objetivo*.

Hay una consideración adicional que hemos de hacer, aunque en principio parezca una obviedad: todo algoritmo bien construido debe finalizar tras la ejecución de un *número finito de pasos*.

Aunque todos los pasos sean de duración finita, una secuencia de instrucciones puede requerir tiempo infinito. Piensa en este método para hacerse millonario:

1) comprar un número de lotería válido para el próximo sorteo,

2) esperar al día de sorteo,

3) cotejar el número ganador con el nuestro,

4) si son diferentes, volver al paso 1; en caso contrario, somos millonarios.

Como ves, cada uno de los pasos del método requiere una cantidad finita de tiempo, pero no hay ninguna garantía de alcanzar el objetivo propuesto.

En adelante, no nos interesarán más las recetas de cocina ni los procedimientos para enriquecerse sin esfuerzo (¡al menos no como objeto de estudio de la asignatura!). Los algoritmos en los que estaremos interesados son aquellos que describen procedimientos de cálculo ejecutables en un ordenador. Ello limitará el ámbito de nuestro estudio a la manipulación y realización de cálculos sobre datos (numéricos, de texto, etc.).

Un algoritmo debe poseer las siguientes características:

1) Ha de tener cero o más *datos de entrada*.

2) Debe proporcionar uno o más *datos de salida* como resultado.

3) Cada paso del algoritmo ha de *estar definido con exactitud*, sin la menor ambigüedad.

4) Ha de ser *finito*, es decir, debe finalizar tras la ejecución de un número finito de pasos, cada uno de los cuales ha de ser ejecutable en tiempo finito.

5) Debe ser *efectivo*, es decir, cada uno de sus pasos ha de poder ejecutarse en tiempo finito con unos recursos determinados (en nuestro caso, con los que proporciona un sistema computador).

Abu Ja'far Mohammed ibn Mûsâ Al-Khowârizm y Euclides

La palabra algoritmo tiene origen en el nombre de un matemático persa del siglo IX: Abu Ja'far Mohammed ibn Mûsâ Al-Khowârizm (que significa «Mohammed, padre de Ja'far, hijo de Moisés, nacido en Khowârizm»). Al-Khowârizm escribió tratados de aritmética y álgebra. Gracias a los textos de Al-Khowârizm se introdujo el sistema de numeración hindú en el mundo árabe y, más tarde, en Occidente.

En el siglo XIII se publicaron los libros *Carmen de Algorismo* (un tratado de aritmética ¡en verso!) y *Algorismus Vulgaris*, basados en parte en la *Aritmética* de Al-Khowârizm. Al-Khowârizm escribió también el libro «Kitab al jabr w'al-muqabala» («Reglas de restauración y reducción»), que dio origen a una palabra que ya conoces: «álgebra».

Abelardo de Bath, uno de los primeros traductores al latín de Al-Khowârizm, empezó un texto con «Dixit Algorismi...» («Dijo Algorismo...»), popularizando así el término *algorismo*, que pasó a significar «realización de cálculos con numerales indoarábigos». En la Edad Media los abaquistas calculaban con ábaco y los algorismistas con «algorismos».

En cualquier caso, el concepto de algoritmo es muy anterior a Al-Khowârizm. En el siglo III a.C., Euclides propuso en su tratado *Elementos* un método sistemático para el cálculo del Máximo Común Divisor (MCD) de dos números. El método, tal cual fue propuesto por Euclides, dice así: «Dados dos números naturales, a y b, comprobar si ambos son iguales. Si es así, a es el MCD. Si no, si a es mayor que b, restar a a el valor de b; pero si a es menor que b, restar a b el valor de a. Repetir el proceso con los nuevos valores de a y b». Este método se conoce como «algoritmo de Euclides», aunque es frecuente encontrar, bajo ese mismo nombre, un procedimiento alternativo y más eficiente: «Dados dos números naturales, a y b, comprobar si b es cero. Si es así, a es el MCD. Si no, calcular c, el resto de dividir a entre b. Sustituir a por b y b por c y repetir el proceso».

Además, nos interesa que los algoritmos sean *eficientes*, esto es, que alcancen su objetivo lo más rápidamente posible y con el menor consumo de recursos.

▶ 9 Diseña un algoritmo para calcular el área de un círculo dado su radio. Recuerda que el área de un círculo es π veces el cuadrado del radio.

No hay una única solución. Diferentes personas ofrecerán algoritmos distintos en función del orden de las operaciones y del nivel de detalle de cada una. Alguien podría proponer este algoritmo breve:

1) Devolver el producto de π por el radio por el radio.

Otro podría alterar el orden de las operaciones:

1) Devolver el producto de radio por π por el radio.

Y otro podría trabajar con pasos más finos:

1) Calcular radio por el radio y denominar «cuadrado» al resultado.

2) Devolver el producto de *pi* por el valor de «cuadrado».

▶ 10 Diseña un algoritmo que calcule el IVA (21 %) de un producto dado su precio de venta sin IVA.

▶ 11 ¿Podemos llamar algoritmo a un procedimiento que escriba en una cinta de papel *todos* los números decimales de π?

Capítulo 2

Una calculadora avanzada

—¿Sabes sumar? —le preguntó la Reina Blanca.— ¿Cuánto es uno más uno más uno más uno más uno más uno más uno más uno más uno más uno más uno?

—No lo sé —dijo Alicia—. Perdí la cuenta.

—No sabe hacer una adición —le interrumpió la Reina Roja.

Alicia en el país de las maravillas, Lewis Carroll

El objetivo de este capítulo es que te familiarices con el *entorno interactivo* de Python, que aprendas a construir *expresiones aritméticas* almacenando los resultados en *variables* mediante *asignaciones* y que conozcas los *tipos de datos* básicos del lenguaje de programación Python.

2.1. Sesiones interactivas

Cuando programamos utilizamos un conjunto de herramientas al que denominamos *entorno de programación*. Entre estas herramientas tenemos editores de texto (que nos permiten escribir programas), compiladores o intérpretes (que traducen los programas a código de máquina), depuradores (que ayudan a detectar errores), analizadores de tiempo de ejecución (para estudiar la eficiencia de los programas), herramientas para la ejecución de pruebas unitarias (para asegurarnos de que no introducimos nuevos errores al ir desarrollando), analizadores de cobertura (para asegurarnos de que todo el código ha sido puesto a prueba), generadores de documentación (que generan documentación en formatos como HTML a partir de comentarios en los programas), analizadores estáticos (que detectan patrones típicos en código erróneo y nos advierten de problemas potenciales), sistemas de control de versiones (que permiten que varios programadores colaboren en un mismo programa, recuperar cualquier versión del programa, crear ramas de desarrollo en paralelo…), etc.

Los lenguajes interpretados suelen ofrecer una herramienta de ejecución *interactiva*. Con ella es posible dar órdenes directamente al intérprete y obtener una respuesta inmediata para cada una de ellas. Es decir, no es necesario escribir un programa completo para empezar a obtener resultados de ejecución, sino que podemos «dialogar» con el intérprete de nuestro lenguaje de programación: le pedimos que ejecute una orden y nos responde con su resultado. El entorno interactivo es de gran ayuda para experimentar con fragmentos de programa antes de incluirlos en una versión definitiva.

En esta sección veremos cómo realizar sesiones de trabajo interactivo con Python[1]. Arrancaremos el intérprete interactivo de modo distinto según el sistema operativo con el que estemos trabajando:

- Si estamos trabajando en un sistema Unix (como cualquiera de las variantes de Linux o Mac OS X), tendremos que iniciar primero un intérprete de órdenes en un terminal. Busca en los menús o barra de aplicaciones algún icono denominado «terminal» o «bash». En la ventana que se abrirá al iniciar el terminal, escribe `python` y pulsa retorno de carro. Si

[1]Abusando del lenguaje, llamaremos indistintamente Python al *entorno de programación*, al *intérprete del lenguaje* y al propio *lenguaje de programación*.

la versión de Python que se ejecuta no es la 3.1 (o superior), sal del intérprete escribiendo `quit()` y pulsando el retorno de carro para, a continuación, escribir `python3` y pulsar nuevamente retorno de carro.

- En Microsoft Windows puedes hacer una de dos cosas:
  - ir al menú de aplicaciones y seleccionar el icono `Python (command line)` en la carpeta `Python 3.x` del menú `Todos los programas`;
  - pulsar las tecla Windows y R simultáneamente para que aparezca un cuadro de diálogo con título `Ejecutar`; escribir en la caja de texto `cmd` y pulsar el botón `Aceptar`; en la ventana que aparece escribe entonces `python` y pulsa el retorno de carro.

El sistema nos responderá dando un mensaje informativo sobre la versión de Python que estamos utilizando (y cuándo fue compilada, con qué compilador, etc.) y, a continuación, mostrará el *prompt*. El resultado será parecido a esto:

```
Python 3.2.3 (default, Feb 27 2014, 21:31:18)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> ↵
```

El *prompt* es la serie de caracteres «>>>» que aparece en la última línea. El *prompt* indica que el intérprete de Python espera que nosotros introduzcamos una orden utilizando el teclado.

Escribamos una expresión aritmética, por ejemplo 2+2, y pulsemos la tecla de retorno de carro. Cuando mostremos sesiones interactivas destacaremos el texto que teclea el usuario con texto de color azul y representaremos con el símbolo ↵ la pulsación de la tecla de retorno de carro. Python *evalúa* la expresión (es decir, obtiene su resultado) y responde mostrando el resultado por pantalla.

```
Python 3.2.3 (default, Feb 27 2014, 21:31:18)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+2↵
4
>>> ↵
```

La última línea es, nuevamente, el *prompt*: Python acabó de ejecutar la última orden (evaluar una expresión y mostrar el resultado) y nos pide que introduzcamos una nueva orden.

Si deseamos acabar la sesión interactiva y salir del intérprete Python, debemos introducir una marca de final de fichero, que en Unix se indica pulsando la tecla de control y, sin soltarla, también la tecla `d`. (De ahora en adelante representaremos una combinación de teclas como la descrita así: `C-d`).

Final de fichero

La «marca de final de fichero» indica que un fichero ha terminado. ¡Pero nosotros no trabajamos con un fichero, sino con el teclado! En realidad, el ordenador considera al teclado como un fichero. Cuando deseamos «cerrar el teclado» para una aplicación, enviamos una marca de final de fichero desde el teclado. En Unix, la marca de final de fichero se envía pulsando la tecla de control y la tecla d simultáneamente, lo que indicamos con `C-d`; en Microsoft Windows, el final de fichero se indica con `C-z`.

Existe otro modo de finalizar la sesión; escribe `quit()` en el intérprete y la sesión se cerrará. En inglés, «quit» significa «abandonar».

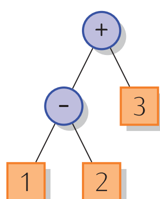
2.1.1. Los operadores aritméticos

Las *operaciones* de suma y resta, por ejemplo, se denotan con los símbolos u *operadores* `+` y `-`, respectivamente, y operan sobre dos valores numéricos (los *operandos*). Probemos algunas expresiones formadas con estos dos operadores:

```
>>> 1 + 2
3
>>> 1 + 2 + 3
6
>>> 1 - 2 + 3
2
```

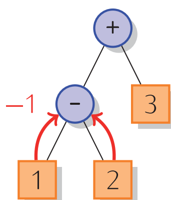
Observa que puedes introducir varias operaciones en una misma línea o expresión. El orden en que se efectúan las operaciones es (en principio) de izquierda a derecha. Por ejemplo, la expresión 1 - 2 + 3 equivale matemáticamente a $((1 - 2) + 3)$; por ello decimos que la suma y la resta son operadores *asociativos por la izquierda*.

Podemos representar gráficamente el orden de aplicación de las operaciones utilizando *árboles sintácticos*. Un árbol sintáctico es una representación gráfica en la que disponemos los operadores y los operandos como nodos y en los que cada operador está conectado a sus operandos. El árbol sintáctico de la expresión «1 - 2 + 3» es este:

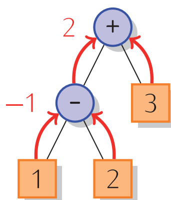


El nodo superior de un árbol recibe el nombre de *nodo raíz*. Los nodos etiquetados con operadores (representados con círculos) se denominan *nodos interiores*. Los nodos interiores tienen uno o más *nodos hijo* o *descendientes* (de los que ellos son sus respectivos *nodos padre* o *ascendientes*). Dichos nodos son nodos raíz de otros (sub)árboles sintácticos (¡la definición de árbol sintáctico es auto-referencial!). Los valores resultantes de evaluar las expresiones asociadas a dichos (sub)árboles constituyen los operandos de la operación que representa el nodo interior. Los nodos sin descendientes se denominan *nodos terminales* u *hojas* (representados con cuadrados) y corresponden a valores numéricos.

La evaluación de cada operación individual en el árbol sintáctico «fluye» de las hojas hacia la raíz (el nodo superior); es decir, en primer lugar se evalúa la subexpresión «1 - 2», que corresponde al subárbol más profundo. El resultado de la evaluación es -1:



A continuación se evalúa la subexpresión que suma el resultado de evaluar «1 - 2» al valor 3:

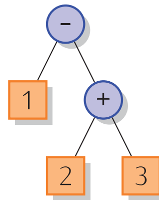


Así se obtiene el resultado final: el valor 2.

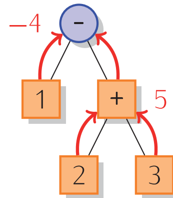
Si deseamos calcular $(1 - (2 + 3))$ podemos hacerlo añadiendo paréntesis a la expresión aritmética:

```
>>> 1 - (2 + 3)
-4
```

El árbol sintáctico de esta nueva expresión es

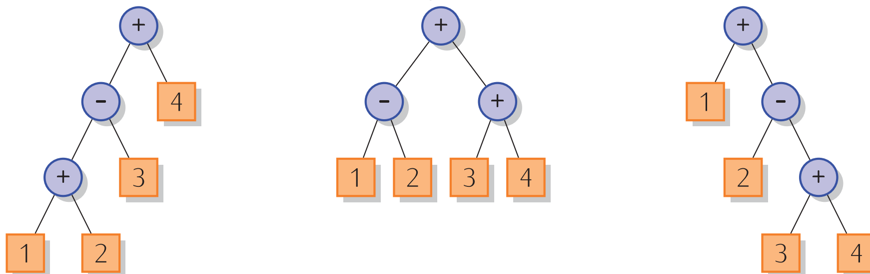


En este nuevo árbol, la primera subexpresión evaluada es la que corresponde al subárbol derecho:



Observa que en el árbol sintáctico no aparecen los paréntesis de la expresión. El árbol sintáctico ya indica el orden en que se procesan las diferentes operaciones y no necesita paréntesis. La expresión Python, sin embargo, *necesita* los paréntesis para indicar ese mismo orden de evaluación.

▶ 12 ¿Qué expresiones Python permiten, utilizando el menor número posible de paréntesis, efectuar *en el mismo orden* los cálculos representados con estos árboles sintácticos?



▶ 13 Dibuja los árboles sintácticos correspondientes a las siguientes expresiones aritméticas:

a) `1 + 2 + 3 + 4`

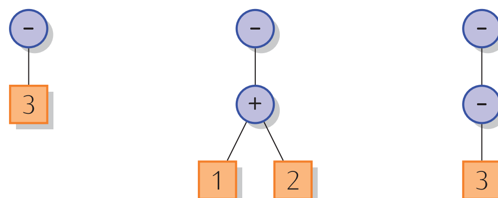
b) `1 - 2 - 3 - 4`

c) `1 - (2 - (3 - 4) + 1)`

Los operadores de suma y resta son *binarios*, es decir, operan sobre dos operandos. El mismo símbolo que se usa para la resta se usa también para un operador *unario*, es decir, un operador que actúa sobre un único operando: el de cambio de signo, que devuelve el valor de su operando cambiado de signo. He aquí algunos ejemplos:

```
>>> -3
-3
>>> -(1 + 2)
-3
>>> -3
3
```

He aquí los árboles sintácticos correspondientes a las tres expresiones del ejemplo:



Espacios en blanco

Parece que se puede hacer un uso bastante liberal de los espacios en blanco en una expresión.

```
>>> 10 + 20 + 30↵
60
>>> 10+20+30↵
60
>>> 10     +20    + 30↵
60
>>> 10+ 20+30↵
60
```

Es así. Has de respetar, no obstante, un par de sencillas reglas. Por una parte no puedes poner espacios en medio de un número:

```
>>> 10 + 2   0 + 30↵
  File "<stdin>", line 1
    10 + 2   0 + 30
             ^
SyntaxError: invalid syntax
```

Los espacios en blanco entre el 2 y el 0 hacen que Python no lea el número 20, sino el número 2 seguido del número 0 (lo cual es un error, pues no hay operación alguna entre ambos números).

Por otra parte, no puedes poner espacios al principio de la expresión:

```
>>>     10 + 20 + 30↵
  File "<stdin>", line 1
    10 + 20 + 30
    ^
IndentationError: unexpected indent
```

Los espacios en blanco entre el *prompt* y el 10 provocan un error. Aún es pronto para que conozcas la razón.

Existe otro operador unario que se representa con el símbolo +: el operador *identidad*. El operador identidad no hace nada «útil»: proporciona como resultado el mismo número que se le pasa.

```
>>> +3↵
3
>>> +-3↵
-3
```

El operador identidad solo sirve para, en ocasiones, poner énfasis en que un número es positivo. (El ordenador considera tan positivo el número 3 como el resultado de evaluar +3).

Los operadores de multiplicación y división son, respectivamente, * y /:

```
>>> 2 * 3↵
6
>>> 3 / 2↵
1.5
>>> 4 / 2↵
2.0
>>> 3 * 4 / 2↵
6.0
>>> 12 / 3 * 2↵
8.0
```

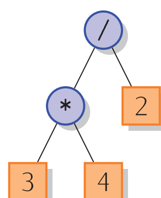
Detengámonos brevemente a hacer una consideración sobre el operador de división. Fíjate en que Python, al dividir 3 entre 2, ha proporcionado como respuesta el valor 1.5. Ese punto que separa el 1 del 5 es lo que en español solemos denotar con una coma[2] y que separa la parte entera de un número de su parte decimal. El operador de división / siempre proporciona un número con parte decimal, aunque esta sea nula; es lo que ocurre al dividir, por ejemplo, 4 entre 2 con el operador /: el resultado es 2.0. Hay otro operador de división que no tiene

[2]El punto también se acepta en español, aunque se prefiere usar la coma. Python representa los números siguiendo el convenio anglosajón.

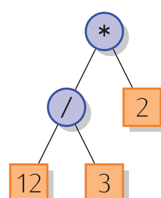
ese efecto: es el operador de división entera // (sin espacio alguno entre las barras). Con // siempre obtienes un número entero como resultado de la división de dos enteros:

```
>>> 3 // 2↵
1
>>> 4 // 2↵
2
>>> -3 // 2↵
-2
```

Observa que los operadores de multiplicación y división (convencional y entera) también son asociativos por la izquierda: la expresión «3 * 4 / 2» equivale a $((3 \cdot 4)/2)$, es decir, tiene el siguiente árbol sintáctico:



y la expresión «12 / 3 * 2» equivale a $((12/3) \cdot 2)$, o sea, su árbol sintáctico es:

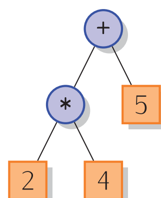


Sigamos estudiando el orden de evaluación cuando una expresión contiene diferentes operadores. ¿Qué pasa si combinamos en una misma expresión operadores de suma o resta con operadores de multiplicación o división? Fíjate en que la regla de aplicación de operadores de izquierda a derecha no siempre se observa:

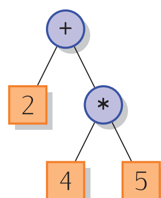
```
>>> 2 * 4 + 5↵
13
>>> 2 + 4 * 5↵
22
```

En la segunda expresión, primero se ha efectuado el producto 4 * 5 y el resultado se ha sumado al valor 2. Ocurre que los operadores de multiplicación y división son *prioritarios* frente a los de suma y resta. Decimos que la multiplicación y la división tienen *mayor nivel de precedencia* o *prioridad* que la suma y la resta.

El árbol sintáctico de 2 * 4 + 5 es:



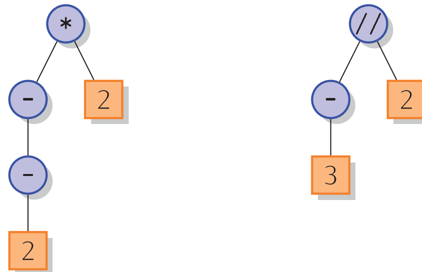
y el de 2 + 4 * 5 es:



Pero ¡atención!, el cambio de signo tiene mayor prioridad que la multiplicación y la división:

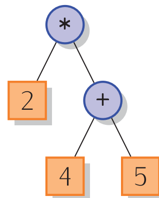
```
>>> -2 * 2
4
>>> -3 // 2
-2
```

Los árboles sintácticos correspondientes a estas dos expresiones son, respectivamente:



Si los operadores siguen unas *reglas de precedencia* que determinan su orden de aplicación, ¿qué hacer cuando deseamos un orden de aplicación distinto? Usar paréntesis, como hacemos con la notación matemática convencional.

La expresión `2 * (4 + 5)`, por ejemplo, presenta este árbol sintáctico:



Existen más operadores en Python. Tenemos, por ejemplo, el operador módulo, que se denota con el símbolo de porcentaje `%` (aunque nada tiene que ver con el cálculo de porcentajes). El operador módulo devuelve el resto de la división entera entre dos operandos.

```
>>> 27 % 5
2
>>> 25 % 5
0
```

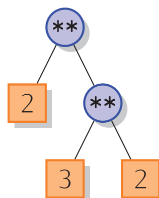
El operador `%` también es asociativo por la izquierda y su prioridad es la misma que la de la multiplicación o la división.

El último operador que vamos a estudiar es la exponenciación, que se denota con dos asteriscos juntos, no separados por ningún espacio en blanco: `**`.

Lo que en notación matemática convencional expresamos como 2^3 se expresa en Python con `2 ** 3`.

```
>>> 2 ** 3
8
```

Pero ¡ojo!, la exponenciación es *asociativa por la derecha*. La expresión `2 ** 3 ** 2` equivale a $2^{(3^2)} = 2^9 = 512$, y no a $(2^3)^2 = 8^2 = 64$, o sea, su árbol sintáctico es:

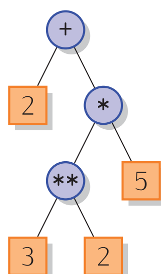


Por otra parte, la exponenciación tiene mayor precedencia que cualquiera de los otros operadores presentados.

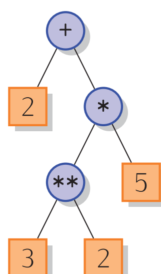
Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

He aquí varias expresiones evaluadas con Python y sus correspondientes árboles sintácticos. Estúdialos con atención:

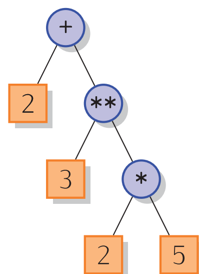
```
>>> 2 + 3 ** 2 * 5↵
47
```



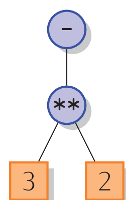

```
>>> 2 + ((3 ** 2) * 5)↵
47
```




```
>>> 2 + 3 ** (2 * 5)↵
59051
```




```
>>> -3 ** 2↵
-9
```



La tabla 2.1 resume las características de los operadores Python que ya conocemos: su aridad (número de operandos), asociatividad y precedencia.

▶ 14 ¿Qué resultados se obtendrán al evaluar las siguientes expresiones Python? Dibuja el árbol sintáctico de cada una de ellas, calcula a mano el valor resultante de cada expresión y comprueba, con la ayuda del ordenador, si tu resultado es correcto.

Operación	Operador	Aridad	Asociatividad	Precedencia
Exponenciación	**	Binario	Por la derecha	1
Identidad	+	Unario	—	2
Cambio de signo	-	Unario	—	2
Multiplicación	*	Binario	Por la izquierda	3
División	/	Binario	Por la izquierda	3
División entera	//	Binario	Por la izquierda	3
Módulo (o resto)	%	Binario	Por la izquierda	3
Suma	+	Binario	Por la izquierda	4
Resta	-	Binario	Por la izquierda	4

Tabla 2.1: Operadores para expresiones aritméticas. El nivel de precedencia 1 es el de mayor prioridad y el 4 el de menor.

a) `2 + 3 + 1 + 2`

b) `2 + 3 * 1 + 2`

c) `(2 + 3) * 1 + 2`

d) `(2 + 3) * (1 + 2)`

e) `+---6`

f) `-+-+6`

g) `-3 / 2 - 1`

h) `-3 // 2 - 1`

▶ 15 Traduce las siguientes expresiones matemáticas a Python y evalúalas. Trata de utilizar el menor número posible de paréntesis.

a) $2 + (3 \cdot (6/2))$

b) $\dfrac{4+6}{2+3}$

c) $(4/2)^5$

d) $(4/2)^{4+2^2}$

e) $(-3)^2$

f) $-(3^2)$

2.1.2. Errores de tecleo y excepciones

Cuando introducimos una expresión y damos la orden de evaluarla, es posible que nos equivoquemos. Si hemos formado incorrectamente una expresión, Python nos lo indicará con un *mensaje de error*. El mensaje de error proporciona información acerca del tipo de error cometido y del lugar en el que este ha sido detectado. Aquí tienes una expresión errónea y el mensaje de error correspondiente:

```
>>> 1 + 2)↵
  File "<stdin>", line 1
    1 + 2)
         ^
SyntaxError: invalid syntax
```


Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

En este ejemplo hemos cerrado un paréntesis cuando no había otro abierto previamente, lo cual es incorrecto. Python nos indica que ha detectado un *error de sintaxis* (`SyntaxError`) y «apunta» con una punta de flecha (el carácter `^`) al lugar en el que se encuentra. (El texto «`File "<stdin>", line 1`» indica que el error se ha producido al leer de teclado, esto es, de la *entrada estándar* —`stdin` es una abreviatura del inglés «*standard input*», que se traduce por «entrada estándar»—).

En Python los errores se denominan *excepciones*. Cuando Python es incapaz de analizar una expresión, produce una excepción. Cuando el intérprete interactivo detecta la excepción, nos muestra por pantalla un mensaje de error.

Veamos algunos otros errores y los mensajes que produce Python.

```
>>> 1 + * 3↵
  File "<stdin>", line 1
    1 + * 3
        ^
SyntaxError: invalid syntax
>>> 2 + 3 %↵
  File "<stdin>", line 1
    2 + 3 %
          ^
SyntaxError: invalid syntax
>>> 1 / 0↵
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: int division or modulo by zero
```

El último error es de naturaleza distinta a los anteriores (no hay un carácter `^` apuntando a lugar alguno): se trata de un *error de división por cero* (`ZeroDivisionError`), cuando los otros eran *errores sintácticos* (`SyntaxError`). La cantidad que resulta de dividir por cero no está definida y Python es incapaz de calcular un valor como resultado de la expresión `1 / 0`. No es un error *sintáctico* porque la expresión está sintácticamente bien formada: el operador de división tiene dos operandos, como toca.

Edición avanzada en el entorno interactivo

Cuando estemos escribiendo una expresión puede que cometamos errores y los detectemos antes de solicitar su evaluación. Aún estaremos a tiempo de corregirlos. La tecla de borrado, por ejemplo, elimina el carácter que se encuentra a la izquierda del cursor. Puedes desplazar el cursor a cualquier punto de la línea que estás editando utilizando las teclas de desplazamiento del cursor a izquierda y a derecha. El texto que teclees se insertará siempre justo a la izquierda del cursor.

Hasta el momento hemos tenido que teclear desde cero cada expresión evaluada, aun cuando muchas se parecían bastante entre sí. Podemos teclear menos si aprendemos a utilizar algunas funciones de edición avanzadas.

Lo primero que hemos de saber es que el intérprete interactivo de Python memoriza cada una de las expresiones evaluadas en una sesión interactiva por si deseamos recuperarlas más tarde. La lista de expresiones que hemos evaluado constituye la *historia* de la sesión interactiva. Puedes «navegar» por la historia utilizando las teclas de desplazamiento del cursor hacia arriba y hacia abajo. Cada vez que pulses la tecla de desplazamiento hacia arriba recuperarás una expresión más antigua. La tecla de desplazamiento hacia abajo permite recuperar expresiones más recientes. La expresión recuperada aparecerá ante el *prompt* y podrás modificarla a tu antojo.

2.2. Tipos de datos

Ya hemos visto que hay dos operadores de división. Uno es el convencional, que siempre produce un número con decimales:

```
>>> 4 / 2↵
2.0
>>> 3 / 2↵
1.5
```

Y otro es el operador de división entera, que siempre produce un número sin decimales cuando sus operandos son enteros:

```
>>> 4 // 2↵
2
>>> 3 // 2↵
1
```

Fíjate en la diferencia entre el resultado de 4 / 2 y 4 // 2. ¿No es lo mismo 2.0 que 2? Pues no exactamente. El número 2 es un número de *tipo entero* y el número 2.0 lo es de *tipo flotante* (o, mejor dicho, *tipo de coma flotante*). Cada valor en Python es una instancia de un *tipo de dato* y de momento hemos visto datos de dos tipos distintos.

2.2.1. Tipos entero y flotante

Hasta el momento hemos utilizado fundamentalmente datos de *tipo entero*, es decir, sin decimales. Solo ocasionalmente hemos visto datos de tipo *flotante*, normalmente como resultado de trabajar con el operador de división convencional. Pero no solo podemos producir datos de tipo flotante con el operador de división convencional: el resto de operadores produce un resultado cuyo tipo depende del tipo de sus operandos. La suma, por ejemplo, produce un resultado de tipo entero si sus dos operandos son de tipo entero, pero produce un valor de tipo flotante si uno cualquiera de sus operandos es de tipo flotante:

```
>>> 2 + 3↵
5
>>> 2.0 + 3↵
5.0
>>> 2 + 3.0↵
5.0
>>> 2.0 + 3.0↵
5.0
```

Python sigue una regla sencilla: si hay datos de tipos distintos, el resultado es del tipo «más general». Los flotantes son de tipo «más general» que los enteros.

```
>>> 1 + 2 + 3 + 4 + 5 + 6 + 0.5↵
21.5
>>> 1 + 2 + 3 + 4 + 5 + 6 + 0.0↵
21.0
```

Hay diferencias entre enteros y reales en Python más allá de que los primeros no tengan decimales y los segundos sí. El número 3 y el número 3.0, por ejemplo, son indistinguibles matemáticamente, pero diferentes en Python. ¿Qué diferencias hay?

- Los enteros suelen ocupar menos memoria.
- Las operaciones entre enteros son, generalmente, más rápidas.

Así pues, utilizaremos enteros a menos que de verdad necesitemos números con decimales.

Hemos de precisar algo respecto a la denominación de los números con decimales: el término «reales» no es adecuado, ya que induce a pensar en los números reales de las matemáticas. En matemáticas, los números reales pueden presentar infinitos decimales, y eso es imposible en un computador. Al trabajar con computadores tendremos que conformarnos con meras *aproximaciones* a los números reales.

Recuerda que todo en el computador son secuencias de ceros y unos. Deberemos, pues, representar internamente con ellos las aproximaciones a los números reales. Para facilitar el intercambio de datos, todos los computadores convencionales utilizan una misma codificación, es decir, representan del mismo modo las aproximaciones a los números reales. Esta codificación se conoce como «IEEE Standard 754 floating point» (que se puede traducir por «Estándar IEEE 754 para coma flotante»), así que llamaremos *números en formato de coma flotante* o simplemente *flotantes* a los números con decimales que podemos representar con el ordenador.

Un número flotante debe especificarse siguiendo ciertas reglas. En principio, consta de dos partes: *mantisa* y *exponente*. El exponente, que debe ser entero, se separa de la *mantisa* con la

letra «e» (o «E»). Por ejemplo, el número flotante 2e3 (o 2E3) tiene mantisa 2 y exponente 3, y representa al número $2 \cdot 10^3$, es decir, 2000.0.

El exponente puede ser negativo: 3.2e-3 es $3.2 \cdot 10^{-3}$, o sea, 0.0032. Ten en cuenta que si un número flotante no lleva exponente *debe* llevar parte fraccionaria.

¡Ah! Un par de reglas más: si la parte entera del número es nula, el flotante puede empezar directamente con un punto, y si la parte fraccionaria es nula, puede acabar con un punto. Veamos un par de ejemplos: el número 0.1 se puede escribir también como .1; por otra parte, el número 2.0 puede escribirse como 2., es decir, en ambos casos el cero es opcional.

¿Demasiadas reglas? No te preocupes, con la práctica acabarás recordándolas.

IEEE Standard 754

Un número en coma flotante presenta tres componentes: el signo, la mantisa y el exponente. He aquí un número en coma flotante: -14.1×10^{-3}. El signo es negativo, la mantisa es 14.1 y el exponente es -3. Los números en coma flotante *normalizada* presentan una mantisa mayor o igual que 1 y menor que 10. El mismo número de antes, en coma flotante normalizada, es -1.41×10^{-2}. Una notación habitual para los números en coma flotante sustituye el producto ($\times$) y la base del exponente por la letra «e» o «E». Notaríamos con -1.41e-2 el número del ejemplo.

Los flotantes de Python siguen la norma IEEE Standard 754. Es una codificación binaria y normalizada de los números en coma flotante y, por tanto, con base 2 para el exponente y mantisa de valor mayor o igual que 1 y menor que 2. Usa 32 bits (precisión simple) o 64 bits (precisión doble) para codificar cada número. Python utiliza el formato de doble precisión. En el formato de precisión doble se reserva 1 bit para el signo del número, 11 para el exponente y 52 para la mantisa. Con este formato pueden representarse números tan próximos a cero como 10^{-323} (322 ceros tras el punto decimal y un uno) o de valor absoluto tan grande como 10^{308}.

No todos los números tienen una representación exacta en el formato de coma flotante. Un número como 0.1 no puede representarse exactamente como flotante. Su mantisa, que vale 1/10, corresponde a la secuencia periódica de bits

0.0001100110011001100110011001100110011001100110011

No hay, pues, forma de representar 1/10 con los 52 bits del formato de doble precisión. En base 10, los 52 primeros bits de la secuencia nos proporcionan el valor

0.1000000000000000055511151231257827021181583404541015625

Es lo más cerca de 1/10 que podemos estar.

Una peculiaridad adicional de los números codificados con la norma IEEE 754 es que su precisión es diferente según el número representado: cuanto más próximo a cero, mayor es la precisión. Para números muy grandes se pierde tanta precisión que no hay decimales (¡ni unidades, ni decenas...!). Por ejemplo, el resultado de la suma 100000000.0+0.000000001 es 100000000.0, y no 100000000.000000001, como cabría esperar.

A modo de conclusión, has de saber que al trabajar con números flotantes es posible que se produzcan pequeños errores en la representación de los valores y durante los cálculos. Probablemente esto te sorprenda, pues es *vox populi* que «los ordenadores nunca se equivocan».

2.2.2. El tipo de datos booleano (y sus operadores)

Hay otro tipo de datos que has de conocer, pues se usa muy frecuentemente en programación: el tipo de datos lógico o booleano, llamado así por ser propio del álgebra de Boole[3]. Un dato de tipo lógico solo puede presentar uno de dos valores: True o False, es decir, verdadero o falso.

Hay tres operadores lógicos en Python: la «*y lógica*» o *conjunción* (**and**), la «*o lógica*» o *disyunción* (**or**) y el «*no lógico*» o *negación* (**not**).

El operador **and** da como resultado True si y solo si son True sus dos operandos. Esta es su *tabla de verdad*:

[3]Boole es el matemático que inventó (¿o descubrió?) el álgebra que lleva su nombre y que se basa en el uso de dos valores (cierto y falso) y tres operadores (negación, conjunción y disyunción).

and		
operandos		**resultado**
izquierdo	derecho	
True	True	True
True	False	False
False	True	False
False	False	False

El operador **or** proporciona **True** si cualquiera de sus operandos es **True**, y **False** solo cuando ambos operandos son **False**. Esta es su tabla de verdad:

or		
operandos		**resultado**
izquierdo	derecho	
True	True	True
True	False	True
False	True	True
False	False	False

El operador **not** es unario, y proporciona el valor **True** si su operando es **False** y viceversa. He aquí su tabla de verdad:

not	
operando	**resultado**
True	False
False	True

Podemos combinar valores lógicos y operadores lógicos para formar *expresiones lógicas*. He aquí algunos ejemplos:

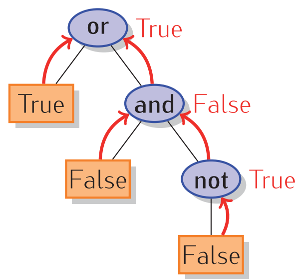
```
>>> True and False↵
False
>>> not True↵
False
>>> (True and False) or True↵
True
>>> True and True or False↵
True
>>> False and True or True↵
True
>>> False and True or False↵
False
```

Has de tener en cuenta la precedencia de los operadores lógicos, que se muestra en la tabla 2.2.

Operación	Operador	Aridad	Asociatividad	Precedencia
Negación	**not**	Unario	—	alta
Conjunción	**and**	Binario	Por la izquierda	media
Disyunción	**or**	Binario	Por la izquierda	baja

Tabla 2.2: Aridad, asociatividad y precedencia de los operadores lógicos.

Del mismo modo que hemos usado árboles sintácticos para entender el proceso de cálculo de los operadores aritméticos sobre valores enteros y flotantes, podemos recurrir a ellos para interpretar el orden de evaluación de las expresiones lógicas. He aquí el árbol sintáctico de la expresión **True or False and not False**:



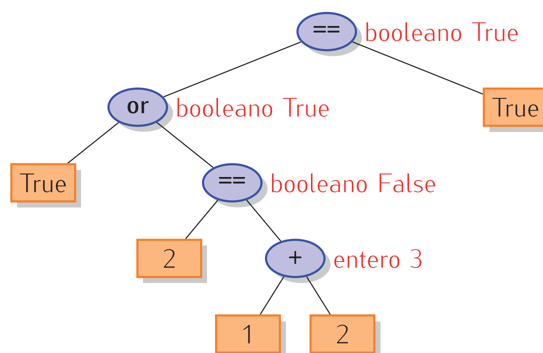
Hay una familia de operadores que devuelven valores booleanos. Entre ellos tenemos a los operadores de comparación, que estudiamos en este apartado. Uno de ellos es el operador de igualdad, que devuelve True si los valores comparados son iguales. El operador de igualdad se denota con dos iguales seguidos: ==. Veámoslo en funcionamiento:

```
>>> 2 == 3↵
False
>>> 2 == 2↵
True
>>> 2.1 == 2.1↵
True
>>> True == True↵
True
>>> True == False↵
False
>>> 2 == 1+1↵
True
```

Observa la última expresión evaluada: es posible combinar operadores de comparación y operadores aritméticos. No solo eso, también podemos combinar en una misma expresión operadores lógicos, aritméticos y de comparación:

```
>>> (True or (2 == 1 + 2)) == True↵
True
```

Este es el árbol sintáctico correspondiente a esa expresión:



Hemos indicado junto a cada nodo interior el tipo del resultado que corresponde a su subárbol. Como ves, en todo momento operamos con tipos compatibles entre sí.

Antes de presentar las reglas de asociatividad y precedencia que son de aplicación al combinar diferentes tipos de operador, te presentamos todos los operadores de comparación en la tabla 2.3 y te mostramos algunos ejemplos de uso:

```
>>> 2 < 1↵
False
>>> 1 < 2↵
True
>>> 5 > 1↵
True
>>> 5 >= 1↵
True
>>> 5 > 5↵
False
```

operador	comparación
!=	es distinto de
==	es igual que
<	es menor que
<=	es menor o igual que
>	es mayor que
>=	es mayor o igual que

Tabla 2.3: Operadores de comparación.

```
>>> 5 >= 5↵
True
>>> 1 != 0↵
True
>>> 1 != 1↵
False
>>> -2 <= 2↵
True
```

Es hora de que presentemos una tabla completa (tabla 2.4) con todos los operadores que conocemos para comparar entre sí la precedencia de cada uno de ellos cuando aparece combinado con otros.

Operación	Operador	Aridad	Asociatividad	Precedencia
Exponenciación	**	Binario	Por la derecha	1
Identidad	+	Unario	—	2
Cambio de signo	-	Unario	—	2
Multiplicación	*	Binario	Por la izquierda	3
División	/	Binario	Por la izquierda	3
División entera	//	Binario	Por la izquierda	3
Módulo (o resto)	%	Binario	Por la izquierda	3
Suma	+	Binario	Por la izquierda	4
Resta	-	Binario	Por la izquierda	4
Igual que	==	Binario	—	5
Distinto de	!=	Binario	—	5
Menor que	<	Binario	—	5
Menor o igual que	<=	Binario	—	5
Mayor que	>	Binario	—	5
Mayor o Igual que	>=	Binario	—	5
Negación	not	Unario	—	6
Conjunción	and	Binario	Por la izquierda	7
Disyunción	or	Binario	Por la izquierda	8

Tabla 2.4: Características de los operadores Python. El nivel de precedencia 1 es el de mayor prioridad.

En la tabla 2.4 hemos omitido cualquier referencia a la asociatividad de los comparadores de Python, pese a que son binarios. Python es un lenguaje peculiar en este sentido. Imaginemos que fueran asociativos por la izquierda. ¿Qué significaría esto? El operador suma, por ejemplo, es asociativo por la izquierda. Al evaluar la expresión aritmética 2 + 3 + 4 se procede así: primero se suma el 2 al 3; a continuación, el 5 resultante se suma al 4 y se obtiene un total de 9.

Si el operador < fuese asociativo por la izquierda, la expresión lógica 2 < 3 < 4 se evaluaría así: primero, se compara el 2 con el 3, resultando el valor True; a continuación, se compara el resultado obtenido con el 4, pero ¿qué significa la expresión True < 4? No tiene sentido.

Cuando aparece una sucesión de comparadores como, por ejemplo, 2 < 3 < 4, Python la evalúa igual que (2 < 3) and (3 < 4). Esta solución permite expresar condiciones complejas de modo sencillo y, en casos como el de este ejemplo, se lee del mismo modo que se leería con la notación matemática habitual, lo cual parece deseable. Pero ¡ojo! Python permite expresiones que son más extrañas; por ejemplo, 2 < 3 > 1, o 2 < 3 == 5.

Una rareza de Python: la asociatividad de los comparadores

Algunos lenguajes de programación de uso común, como C y C++, hacen que sus operadores de comparación sean asociativos, por lo que presentan el problema de que expresiones como `2 < 1 < 4` producen un resultado que parece ilógico. Al ser asociativo por la izquierda el operador de comparación `<`, se evalúa primero la subexpresión `2 < 1`. El resultado es *falso*, que en C y C++ se representa con el valor 0. A continuación se evalúa la comparación `0 < 4`, cuyo resultado es... ¡cierto! Así pues, para C y C++ es cierto que `2 < 1 < 4`.

Pascal es más rígido aún y llega a prohibir expresiones como `2 < 1 < 4`. En Pascal hay un tipo de datos denominado **`boolean`** cuyos valores válidos son **`true`** y **`false`**. Pascal no permite operar entre valores de tipos diferentes, así que la expresión `2 < 1` se evalúa al valor booleano **`false`**, que no se puede comparar con un entero al tratar de calcular el valor de **`false`** `< 4`. En consecuencia, se produce un error de tipos si intentamos encadenar comparaciones.

La mayor parte de los lenguajes de programación convencionales opta por la solución del C o por la solución del Pascal. Cuando aprendas otro lenguaje de programación, te costará «deshabituarte» de la elegancia con que Python resuelve los encadenamientos de comparaciones.

▶ 16 ¿Qué resultados se muestran al evaluar estas expresiones?

```
>>> True == True != False↵
>>> 1 < 2 < 3 < 4 < 5↵
>>> (1 < 2 < 3) and (4 < 5)↵
>>> 1 < 2 < 4 < 3 < 5↵
>>> (1 < 2 < 4) and (3 < 5)↵
```

2.3. Literales de entero

Ya sabes cómo escribir un número entero... en base 10. Es lo más corriente: escribir números enteros con un sistema en el que cada dígito es un número entre 0 y 9, y en el que cada posición supone que el dígito vale 10 veces más que el que hay más a su derecha. Así, el número 132 equivale a $1 \cdot 10^2 + 3 \cdot 10^1 + 2 \cdot 10^0 = 1 \cdot 100 + 3 \cdot 10 + 2 \cdot 1$. Pero en ocasiones convendrá expresar números con otra base. En informática es muy común trabajar con base 2, base 8 y base 16. Es natural que la base 2 sea usada con relativa frecuencia, pues ya sabes que la información se almacena y manipula a partir de codificaciones binarias. Las otras dos bases son de uso común por razones históricas. La base 8 permite manejar números en los que interpretamos que cada grupo de 3 bits corresponde a un dígito y la base 16 hace lo propio con grupos de 4 bits.

Python permite expresar números en estas tres bases. Los números en base 2 se expresan con el prefijo 0b (o 0B) seguido de una sucesión de unos y ceros.

```
>>> 0b1 + 0b0001↵
2
>>> 0b10 * 0b110↵
12
```

Observa que el resultado se muestra en base 10, aunque los números se hayan escrito en base 2. Tan pronto Python analiza la secuencia 0b10, por ejemplo, considera que ha leído el valor 2. Internamente no hay ninguna diferencia entre el resultado de evaluar 2 y 0b10. Así pues, el resultado de evaluar la expresión 0b1 + 0b0001 es exactamente el mismo que obtenemos al evaluar 1+1.

Los números en base 8, también conocida como base octal, se forman con 0o (o 0O) seguida de uno o más dígitos entre 0 y 7. El número 0o177 corresponde al número decimal $1 \cdot 8^2 + 7 \cdot 8^1 + 7 \cdot 8^0 = 1 \cdot 64 + 7 \cdot 8 + 7 \cdot 1 = 64 + 56 + 7 = 127$.

```
>>> 0o10 + 0o10↵
16
>>> 0o7 + 0o177↵
134
```

Finalmente, los números en base 16 o hexadecimal empiezan por 0x (o 0X) y siguen con uno o más dígitos entre 0 y 9 o letras minúsculas o mayúsculas entre A y F. La letra A tiene valor 10, la B valor 11 y así hasta la F, que tiene valor 15. El número 0xFA1 corresponde a $15 \cdot 16^2 + 10 \cdot 16^1 + 1 \cdot 16^0 = 15 \cdot 256 + 10 \cdot 16 + 1 \cdot 1 = 4001$.

```
>>> 0xff + 0x1↵
256
>>> 0xff * 0x2↵
510
```

¿Y si quisiésemos ver el resultado en base 2, 8 o 16? Hemos de aprender antes algunas cosas acerca de las cadenas y las funciones. Antes de acabar este capítulo sabremos cómo hacerlo.

▶ 17 Evalúa estas expresiones:

a) 0xf + 0o17 + 0b1111 + 15

b) 0xffff + 0b1

2.4. Variables y asignaciones

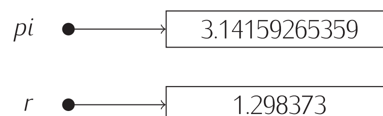
En ocasiones deseamos que el ordenador recuerde ciertos valores para usarlos más adelante. Por ejemplo, supongamos que deseamos efectuar el cálculo del perímetro y el área de un círculo de radio 1.298373 m. La fórmula del perímetro es $2\pi r$, donde r es el radio, y la fórmula del área es πr^2. (Aproximaremos el valor de π con 3.14159265359). Podemos realizar ambos cálculos del siguiente modo:

```
>>> 2 * 3.14159265359 * 1.298373↵
8.157918156839218
>>> 3.14159265359 * 1.298373 ** 2↵
5.296010335524904
```

Observa que hemos tenido que introducir dos veces los valores de π y r por lo que, al tener tantos decimales, es muy fácil cometer errores. Para paliar este problema podemos utilizar *variables*:

```
>>> pi = 3.14159265359↵
>>> r = 1.298373↵
>>> 2 * pi * r↵
8.157918156839218
>>> pi * r ** 2↵
5.296010335524904
```

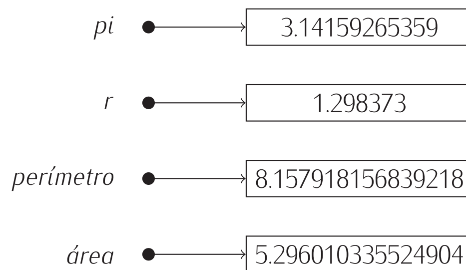
En la primera línea hemos creado una variable de nombre *pi* y valor 3.14159265359. A continuación, hemos creado otra variable, *r*, y le hemos dado el valor 1.298373. El acto de dar valor a una variable se denomina *asignación*. Al asignar un valor a una variable que no existía, Python reserva un espacio en la memoria, almacena el valor en él y crea una asociación entre el nombre de la variable y la dirección de memoria de dicho espacio. Podemos representar gráficamente el resultado de estas acciones así:



A partir de ese instante, escribir *pi* es equivalente a escribir 3.14159265359, y escribir *r* es equivalente a escribir 1.298373.

Podemos almacenar el resultado de calcular el perímetro y el área en sendas variables:

```
>>> pi = 3.14159265359↵
>>> r = 1.298373↵
>>> perímetro = 2 * pi * r↵
>>> área = pi * r**2↵
```



La memoria se ha reservado correctamente, en ella se ha almacenado el valor correspondiente y la asociación entre la memoria y el nombre de la variable se ha establecido, pero no obtenemos respuesta alguna por pantalla. Debes tener en cuenta que las asignaciones son «mudas», es decir, no provocan salida por pantalla. Si deseamos ver cuánto vale una variable, podemos evaluar una expresión que solo contiene a dicha variable:

```
>>> pi = 3.14159265359↵
>>> r = 1.298373↵
>>> perímetro = 2 * pi * r↵
>>> área = pi * r**2↵
>>> perímetro↵
8.157918156839218
>>> área↵
5.296010335524904
```

Así pues, para asignar valor a una variable basta ejecutar una sentencia como esta:

variable = *expresión*

Ten cuidado: el orden es importante. Hacer «*expresión* = *variable*» no es equivalente. *Una asignación no es una ecuación matemática*, sino una acción consistente en (por este orden):

1) evaluar la expresión *a la derecha* del símbolo igual (=), y

2) guardar el valor resultante en la variable indicada *a la izquierda* del símbolo igual.

== no es = (comparar no es asignar)

Al aprender a programar, muchas personas confunden el operador de asignación, =, con el operador de comparación, ==. El primero se usa exclusivamente para asignar un valor a una variable. El segundo, para comparar valores.

Observa la diferente respuesta que obtienes al usar = y == en el entorno interactivo:

```
>>> a = 10↵
>>> a↵
10
>>> a == 1↵
False
>>> a↵
10
```

Se puede asignar valor a una misma variable cuantas veces se quiera. El efecto es que la variable, en cada instante, solo «recuerda» el último valor asignado... hasta que se le asigne otro.

```
>>> a = 1
>>> 2 * a
2
>>> a + 2
3
>>> a = 2
>>> a * a
4
```

Una asignación no es una ecuación

Hemos de insistir en que las asignaciones no son ecuaciones matemáticas, por mucho que su aspecto nos recuerde a estas. Fíjate en este ejemplo, que suele sorprender a aquellos que empiezan a programar:

```
>>> x = 3
>>> x = x + 1
>>> x
4
```

La primera línea asigna a la variable x el valor 3. La segunda línea parece más complicada. Si la interpretas como una ecuación, no tiene sentido, pues de ella se concluye absurdamente que $3 = 4$ o, sustrayendo la x a ambos lados del igual, que $0 = 1$. Pero si seguimos paso a paso las acciones que ejecuta Python al hacer una asignación, la cosa cambia:

1) Se evalúa la parte derecha del igual (sin tener en cuenta para nada la parte izquierda). El valor de x es 3, que sumado a 1 da 4.
2) El resultado (el 4), se almacena en la variable que aparece en la parte izquierda del igual, es decir, en x.

Así pues, el resultado de ejecutar las dos primeras líneas es que x vale 4.

El nombre de una variable es su *identificador*. Hay unas reglas precisas para construir identificadores. Si no se siguen, diremos que el identificador no es válido. Un identificador debe estar formado por letras[4] minúsculas, mayúsculas, dígitos y/o el carácter de subrayado _, con una restricción: que el primer carácter no sea un dígito.

Hay una norma más: un identificador no puede coincidir con una *palabra reservada* o *palabra clave*. Una palabra reservada es una palabra que tiene un significado predefinido y es necesaria para expresar ciertas construcciones del lenguaje. Aquí tienes una lista con todas las palabras reservadas de Python: **and**, **as**, **assert**, **break**, **class**, **continue**, **def**, **del**, **elif**, **else**, **except**, **False**, **finally**, **for**, **from**, **global**, **if**, **import**, **in**, **is**, **lambda**, **nonlocal**, **None**, **not**, **or**, **pass**, **raise**, **return**, **True**, **try**, **with**, **while** y **yield**.

Por ejemplo, los siguientes identificadores son válidos: *h*, *x*, *Z*, *velocidad*, *aceleración*, *fuerza1*, *masa_2*, *_a*, *a_*, *prueba_123*, *desviación_típica*. Debes tener presente que Python distingue entre mayúsculas y minúsculas, así que *área*, *Área* y *ÁREA* son tres identificadores válidos y diferentes.

Cualquier carácter diferente de una letra, un dígito o el subrayado es inválido en un identificador, incluyendo el espacio en blanco. Por ejemplo, *edad media* (con un espacio en medio) son *dos* identificadores (*edad* y *media*), no uno. Cuando un identificador se forma con dos palabras, es costumbre de muchos programadores usar el subrayado para separarlas: *edad_media*; otros programadores utilizan una letra mayúscula para la inicial de la segunda: *edadMedia*. Escoge el estilo que más te guste para nombrar variables, pero permanece fiel al que escojas.

Dado que eres libre de llamar a una variable con el identificador que quieras, hazlo con clase: *escoge siempre nombres que guarden relación con los datos del problema*. Si, por ejemplo, vas a utilizar una variable para almacenar una distancia, llama a la variable *distancia* y evita nombres que no signifiquen nada; de este modo, los programas serán más legibles.

▶ 18 ¿Son válidos los siguientes identificadores?

a) *Identificador*

[4] Son letras válidas las de cualquier alfabeto. Así, π es un identificador válido por ser una letra del alfabeto griego.

b) *Indice\dos*

c) *Dos palabras*

d) __

e) 12*horas*

f) *hora12*

g) *desviación*

h) *año*

i) *from*

j) *var*!

k) `'var'`

l) *import_from*

m) *UnaVariable*

n) *a*(*b*)

ñ) 12

o) *uno*.*dos*

p) *x*

q) π

r) *área*

s) *area*-*rect*

t) *x*________*1*

u) _________*1*

v) _*x*_

w) *x*_*x*

▶ 19 ¿Qué resulta de ejecutar estas tres líneas?

```
>>> x = 10↲
>>> x = x * 10↲
>>> x↲
```

▶ 20 Evalúa el polinomio $x^4 + x^3 + 2x^2 - x$ en $x = 1.1$. Utiliza variables para evitar teclear varias veces el valor de x. (El resultado es 4.1151).

▶ 21 Evalúa el polinomio $x^4 + x^3 + \frac{1}{2}x^2 - x$ en $x = 10$. (El resultado es 11040.0).

2.4.1. Asignaciones con operador

Fíjate en la sentencia $i = i + 1$: aplica un incremento unitario al contenido de la variable i. Incrementar el valor de una variable en una cantidad cualquiera es tan frecuente que existe una forma compacta en Python. El incremento de i puede denotarse así: i += 1 (sin espacio alguno entre el + y el =). Puedes incrementar una variable con cualquier cantidad, incluso con una que resulte de evaluar una expresión:

```
>>> a = 3↵
>>> b = 2↵
>>> a += 4 * b↵
>>> a↵
11
```

Todos los operadores aritméticos tienen su asignación con operador asociada.

```
>>> z = 1↵
>>> z += 2↵
>>> z *= 2↵
>>> z //= 2↵
>>> z -= 2↵
>>> z %= 2↵
>>> z **= 2↵
>>> z /= 2↵
>>> z↵
0.5
```

Hemos de decirte que estas formas compactas no aportan nada nuevo... salvo comodidad, así que no te preocupes por tener que aprender tantas cosas. Si te vas a sentir incómodo por tener que tomar decisiones y siempre estás pensando «¿uso ahora la forma normal o la compacta?», será mejor que ignores de momento las formas compactas.

▶ 22 ¿Qué valor tiene z tras evaluar estas sentencias?

```
>>> z = 2↵
>>> z += 2↵
>>> z += 2 - 2↵
>>> z *= 2↵
>>> z *= 1 + 1↵
>>> z //= 2↵
>>> z %= 3↵
>>> z /= 3 - 1↵
>>> z -= 2 + 1↵
>>> z -= 2↵
>>> z **= 3↵
>>> z↵
```

2.4.2. Variables no inicializadas

En Python, la primera operación sobre una variable debe ser la asignación de un valor. No se puede usar una variable a la que no se ha asignado previamente un valor:

```
>>> a + 2↵
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'a' is not defined
```

Como puedes ver, se genera una excepción `NameError`, es decir, de «error de nombre». El texto explicativo precisa aún más lo sucedido: «`name 'a' is not defined`», es decir, «el nombre a no está definido».

La asignación de un valor inicial a una variable se denomina *inicialización* de la variable. Decimos, pues, que en Python no es posible usar variables no inicializadas.

Más operadores

Solo te hemos presentado los operadores que utilizaremos en el texto y que ya estás preparado para manejar. Pero has de saber que hay más operadores. Hay operadores, por ejemplo, que están dirigidos a manejar las secuencias de bits que codifican los valores enteros. El operador binario `&` calcula la operación «y» bit a bit, el operador binario `|` calcula la operación «o» bit a bit, el operador binario `^` calcula la «o exclusiva» (que devuelve cierto si y solo si los dos operandos son distintos), también bit a bit, y el operador unario `~` invierte los bits de su operando. Tienes, además, los operadores binarios `<<` y `>>`, que desplazan los bits a izquierda o derecha tantas posiciones como le indiques. Estos ejemplos te ayudarán a entender estos operadores:

En decimal		En binario	
Expresión	Resultado	Expresión	Resultado
5 & 12	4	00000101 & 00001100	00000100
5 | 12	13	00000101 | 00001100	00001101
5 ^ 12	9	00000101 ^ 00001100	00001001
5 << 1	10	00000101 << 00000001	00001010
5 << 2	20	00000101 << 00000010	00010100
5 << 3	40	00000101 << 00000011	00101000
5 >> 1	2	00000101 >> 00000001	00000010

¡Y estos operadores presentan, además, una forma compacta con asignación: `<<=`, `|=`, etc.!

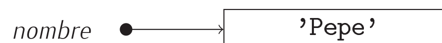
Más adelante estudiaremos, además, los operadores `is` (e `is not`) e `in` (y `not in`), los operadores de indexación, de llamada a función, de corte...

2.5. El tipo de datos cadena

Hasta el momento hemos visto que Python puede manipular datos numéricos de dos tipos: enteros y flotantes. Pero Python también puede manipular otros tipos de datos. Vamos a estudiar ahora el tipo de datos que se denomina *cadena*. Una cadena es una secuencia de caracteres (letras, números, espacios, marcas de puntuación, etc.) y en Python se distingue porque va *encerrada entre comillas simples o dobles* . Por ejemplo, `'cadena'`, `'otro␣ejemplo'`, `"1,␣2␣1o␣3"`, `'¡Si!'`, `"...Python"` son cadenas. Observa que los espacios en blanco se muestran así en este texto: «␣». Lo hacemos para que resulte fácil contar los espacios en blanco cuando haya más de uno seguido. Esta cadena, por ejemplo, está formada por tres espacios en blanco: `'␣␣␣'`. Si no los representásemos con las cajitas, sería difícil contarlos.

Las cadenas pueden usarse para representar información textual: nombres de personas, nombres de colores, matrículas de coche... Las cadenas también pueden almacenarse en variables.

```
>>> nombre = 'Pepe'↵
>>> nombre↵
'Pepe'
```



Es posible realizar operaciones con cadenas. Por ejemplo, podemos «sumar» cadenas añadiendo una a otra.

```
>>> 'a' + 'b'↵
'ab'
>>> nombre = 'Pepe'↵
>>> nombre + 'Cano'↵
'PepeCano'
>>> nombre + '␣' + 'Cano'↵
'Pepe␣Cano'
>>> apellido = 'Cano'↵
>>> nombre + '␣' + apellido↵
'Pepe␣Cano'
```

Una cadena no es un identificador

Con las cadenas tenemos un problema: muchas personas que están aprendiendo a programar confunden una cadena con un identificador de variable y viceversa. No son la misma cosa. Fíjate bien en lo que ocurre:

```
>>> a = 1↵
>>> 'a'↵
'a'
>>> a↵
1
```

La primera línea asigna a la variable *a* el valor 1. Como *a* es el nombre de una variable, es decir, un identificador, *no va encerrado entre comillas*. A continuación hemos escrito **'a'** y Python ha respondido también con **'a'**: la *a* entre comillas es una cadena formada por un único carácter, la letra «a», y no tiene *nada* que ver con la variable *a*. A continuación hemos escrito la letra «a» sin comillas y Python ha respondido con el valor 1, que es lo que contiene la variable *a*.

Muchos estudiantes de programación cometen errores como estos:

- Quieren utilizar una cadena, pero olvidan las comillas, con lo que Python cree que se quiere usar un identificador; si ese identificador no existe, da un error:

```
>>> Pepe↵
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'Pepe' is not defined
```

- Quieren usar un identificador pero, ante la duda, lo encierran entre comillas:

```
>>> 'x' = 2↵
  File "<input>", line 1
SyntaxError: can't assign to literal
```

Recuerda: solo se puede asignar valores a variables, nunca a cadenas, y las cadenas no son identificadores.

Hablando con propiedad, esta operación no se llama suma, sino *concatenación*. El símbolo utilizado es +, el mismo que usamos cuando sumamos enteros y/o flotantes; pero aunque el símbolo sea el mismo, ten en cuenta que no es igual sumar números que concatenar cadenas:

```
>>> '12' + '12'↵
'1212'
>>> 12 + 12↵
24
```

Sumar o concatenar una cadena y un valor numérico (entero o flotante) produce un error:

```
>>> '12' + 12↵
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Y para acabar, hay un operador de repetición de cadenas. El símbolo que lo denota es *, el mismo que hemos usado para multiplicar enteros y/o flotantes. El operador de repetición necesita dos datos: uno de tipo cadena y otro de tipo entero. El resultado es la concatenación de la cadena consigo misma tantas veces como indique el número entero:

```
>>> 'Hola' * 5↵
'HolaHolaHolaHolaHola'
>>> '-' * 60↵
'------------------------------------------------------------'
>>> 60 * '-'↵
'------------------------------------------------------------'
```

▶ 23 Evalúa estas expresiones y sentencias en el mismo orden en el que aparecen e indica lo que muestra el intérprete de Python como respuesta.

```
>>> a = 'b'↵
>>> a + 'b'↵
>>> a + 'a'↵
>>> a * 2 + 'b' * 3↵
>>> 2 * (a + 'b')↵
>>> 2 * ('a' + 'b')↵
```

▶ 24 ¿Qué resultados se obtendrán al evaluar las siguientes expresiones y asignaciones Python? Calcula primero a mano el valor resultante de cada expresión y comprueba, con la ayuda del ordenador, si tu resultado es correcto.

```
>>> 'a' * 3 + '/*' * 5 + 2 * 'abc' + '+'↵
>>> palindromo = 'abcba'↵
>>> (4 * '<' + palindromo + '>' * 4) * 2↵
>>> subcadena = '=' + '-' * 3 + '='↵
>>> '10' * 5 + 4 * subcadena↵
>>> 2 * '12' + '.' + '3' * 3 + 'e-' + 4 * '76'↵
```

▶ 25 Identifica regularidades en las siguientes cadenas, y escribe expresiones que, partiendo de subcadenas más cortas y utilizando los operadores de concatenación y repetición, produzcan las cadenas que se muestran. Introduce variables para formar las expresiones cuando lo consideres oportuno.

a) `'%%%%%./././<-><->'`

b) `'(@)(@)(@)======(@)(@)(@)======'`

c) `'asdfasdfasdf=-=-=-=-=-=-=-??????asdfasdf'`

d) `'........*****---*****---........*****---*****---'`

2.6. Funciones predefinidas

Hemos estudiado los operadores aritméticos básicos. Python también proporciona funciones que podemos utilizar en las expresiones. Estas funciones se dice que están *predefinidas*[5].

2.6.1. Algunas funciones sobre valores numéricos

La función *abs*, por ejemplo, calcula el valor absoluto de un número. Podemos usarla como en estas expresiones:

```
>>> abs(-3)↵
3
>>> abs(3)↵
3
```

El número sobre el que se aplica la función se denomina *argumento*. Observa que el argumento de la función debe ir encerrado entre paréntesis:

```
>>> abs(0)↵
0
>>> abs 0↵
  File "<input>", line 1
    abs 0
        ^
SyntaxError: invalid syntax
```

Existen muchas funciones predefinidas, pero es pronto para aprenderlas todas. Te resumimos algunas que ya puedes utilizar:

[5] *Pre*definidas porque nosotros también podemos definir nuestras propias funciones. Ya llegaremos.

- *float*: conversión a flotante. Si recibe un número entero como argumento, devuelve el mismo número convertido en un flotante equivalente.

```
>>> float(3)↵
3.0
```

La función *float* también acepta argumentos de tipo cadena. Cuando se le pasa una cadena, *float* la convierte en el número flotante que esta representa:

```
>>> float('3.2')↵
3.2
>>> float('3.2e10')↵
32000000000.0
```

Pero si la cadena no representa un flotante, se produce un error de tipo `ValueError`, es decir, «error de valor»:

```
>>> float('un␣texto')↵
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: could not convert string to float: 'un␣texto'
```

Si *float* recibe un argumento flotante, devuelve el mismo valor que se suministra como argumento.

- *int*: conversión a entero. Si recibe un número flotante como argumento, devuelve el entero que se obtiene eliminando la parte fraccionaria.[6]

```
>>> int(2.1)↵
2
>>> int(-2.9)↵
-2
```

La función *int* acepta como argumento una cadena:

```
>>> int('2')↵
2
```

Si *int* recibe un argumento entero, devuelve un entero con el valor del argumento, tal cual.

- *str*: conversión a cadena. Recibe un número y devuelve una representación de este como cadena.

```
>>> str(2.1)↵
'2.1'
>>> str(234E47)↵
'2.34e+49'
```

La función *str* también puede recibir como argumento una cadena, pero en ese caso devuelve como resultado la misma cadena.

[6]El redondeo de *int* puede ser al alza o a la baja según el ordenador en que lo ejecutes. Esto es así porque *int* se apoya en el comportamiento del redondeo automático de C (el intérprete de Python que usamos está escrito en C) y su comportamiento está indefinido. Si quieres un comportamiento homogéneo del redondeo, puedes usar las funciones *round*, *floor* o *ceil*, que se explican más adelante.

- *bin*: representación en binario. Convierte un número entero en una cadena con el número expresado en base 2.

```
>>> bin(3)↵
'0b11'
>>> bin(10)↵
'0b1010'
>>> bin(254)↵
'0b11111110'
```

- *oct*: representación en octal. Convierte un número entero en una cadena con el número expresado en base 8.

```
>>> oct(3)↵
'0o3'
>>> oct(10)↵
'0o12'
>>> oct(254)↵
'0o376'
```

- *hex*: representación en hexadecimal. Convierte un número entero en una cadena con el número expresado en base 16.

```
>>> hex(3)↵
'0x3'
>>> hex(10)↵
'0xa'
>>> hex(254)↵
'0xfe'
```

- *round*: redondeo. Puede usarse con uno o dos argumentos. Si se usa con un solo argumento, redondea el número al entero más próximo.

```
>>> round(2.1)↵
2
>>> round(2.9)↵
3
>>> round(-2.9)↵
-3
>>> round(2)↵
2
```

(¡Observa que el resultado siempre es de tipo entero!) Si *round* recibe dos argumentos, *estos deben ir separados por una coma* y el segundo indica el número de decimales que deseamos conservar tras el redondeo.

```
>>> round(2.1451, 2)↵
2.15
>>> round(2.1451, 3)↵
2.145
>>> round(2.1451, 0)↵
2.0
```

Estas funciones (y las que estudiaremos más adelante) pueden formar parte de expresiones y sus argumentos pueden, a su vez, ser expresiones. Observa los siguientes ejemplos:

```
>>> abs(-23) % int(7.3)↵
2
>>> abs(round(-34.2765, 1))↵
34.3
>>> str(float(str(2) * 3 + '.123')) + '321'↵
'222.123321'
```

▶ 26 Calcula con una única expresión el valor absoluto del redondeo de −3.2. (El resultado es 3).

▶ 27 Convierte (en una única expresión) a una cadena el resultado de la división 5011/10000 redondeado con 3 decimales.

▶ 28 ¿Qué resulta de evaluar estas expresiones?

```
>>> str(2.1) + str(1.2)↵
>>> int(str(2) + str(3))↵
>>> str(int(12.3)) + '0'↵
>>> int('2'+'3')↵
>>> str(2 + 3)↵
>>> str(int(2.1) + float(3))↵
```

2.6.2. Dos funciones básicas para cadenas: *ord* y *chr*

El concepto de comparación entre números te resulta familiar porque lo has estudiado antes en matemáticas. Python extiende el concepto de comparación a otros tipos de datos, como las cadenas. En el caso de los operadores `==` y `!=` el significado está claro: dos cadenas son iguales si son iguales carácter a carácter, y distintas en caso contrario. Pero ¿qué significa que una cadena sea menor que otra? Python utiliza un criterio de comparación de cadenas similar al orden alfabético.

Cuando Python compara dos cadenas lo hace carácter a carácter. Cada carácter es un entero de 16 bits. La letra a, por ejemplo, se codifica con el entero 97, y la b con el entero 98. Si comparamos la cadena `'a'` con la cadena `'b'`, Python nos dirá que la primera es menor que la segunda. ¿Qué pasa si comparamos `'aa'` con `'ab'`? Python compara primero el primer carácter de cada cadena. Como los dos son iguales, pasa a comparar el segundo carácter de cada cadena y llega a la conclusión de que la primera cadena es menor que la segunda. ¿Y qué pasa si comparamos `'a'` con `'aa'`? Nuevamente el primer carácter resulta insuficiente para decidir nada. Python trata de pasar a estudiar el segundo carácter de cada cadena, pero la primera cadena no tiene segundo carácter. Así pues, nuevamente resulta que la primera cadena es menor que la segunda.

En principio, una cadena es menor que otra si la debe preceder al disponerlas en un diccionario. Por ejemplo, `'abajo'` es menor que `'arriba'`. Pero solo en principio. Fíjate en que la letra b mayúscula tiene código 66. Eso significa que 'Barco' es menor que 'ancla':

```
>>> 'Barco' < 'ancla'↵
True
```

Las letras acentuadas plantean problemas similares, pues tienen valores numéricos mayores que sus versiones sin acentuar:

```
>>> 'ábaco' < 'ajo'↵
False
```

Para conocer el valor numérico que corresponde a un carácter, puedes utilizar la función predefinida *ord*, a la que le has de pasar el carácter en cuestión como argumento.

```
>>> ord('a')↵
97
>>> ord('A')↵
65
```

De ASCII a Unicode

Hace algunos años era corriente codificar cada carácter con un byte (ocho bits). La tabla que establecía la correspondencia entre carácter y valor numérico era la denominada tabla ASCII, de la que hemos hablado brevemente en el primer capítulo. La letra a, por ejemplo, tiene valor numérico 97. En realidad no codificaba 256 símbolos, sino solo 128: los que correspondían a valores numéricos entre 0 y 127. Esta tabla, diseñada en 1968, era problemática en países como el nuestro, pues no recogía los caracteres acentuados o la letra eñe. Con 256 caracteres, que es lo que podemos codificar con 8 bits, era imposible tener un juego completo válido para todos los países del mundo. De hecho, ni siquiera para todos los países europeos.

Cada sistema informático extendió la tabla ASCII a su gusto. Usualmente se añadían caracteres a los 128 valores no usados por la tabla ASCII. Esto trajo multitud de problemas a la hora de intercambiar archivos de texto entre sistemas. En los años 90, una comisión estandarizó tablas adaptadas a diferentes dominios lingüísticos. La tabla apropiada para Europa occidental era la denominada ISO-8859-1, también conocida como IsoLatin1 o Latin1. Pronto esta tabla se quedó corta: el símbolo del euro no estaba contemplado en ella. La tabla ISO-8859-15 ampliaba la ISO-8859-1 para recoger el símbolo del euro.

El problema de codificar la información textual estaba lejos de quedar satisfactoriamente resuelto si había que recurrir a multitud de tablas de 256 caracteres. Piénsese en que era imposible, por ejemplo, incluir en un único fichero de texto un fragmento en español con otro en japonés.

Surgió entonces una codificación capaz de resolver el problema definitivamente: la codificación Unicode. Unicode empezó planteando que cada carácter debía codificarse con 16 bits y no con solo 8. Esto hacía que hubiera 65536 códigos disponibles. Como seguían siendo insuficientes para representar cualquier carácter de cualquier lengua, Unicode definió codificaciones con número de bits variable que permitieran, mediante sucesivas extensiones, dar cuenta de cualquier alfabeto existente.

Nosotros consideraremos que cada carácter se representa con un número de 16 bits y no entraremos en más detalles sobre Unicode. Python, desde las versiones 3.0 en adelante, representa las cadenas con Unicode.

```
>>> ord('á')↵
225
```

La función *chr* hace lo contrario: devuelve un carácter, dado su valor numérico.

```
>>> chr(97)↵
'a'
>>> chr(65)↵
'A'
```

▶ 29 ¿Qué resultados se muestran al evaluar estas expresiones?

```
>>> 'abalorio' < 'abecedario'↵
>>> 'abecedario' < 'abecedario'↵
>>> 'abecedario' <= 'abecedario'↵
>>> 'Abecedario' < 'abecedario'↵
>>> 'Abecedario' == 'abecedario'↵
>>> 124 < 13↵
>>> '124' < '13'↵
>>> '␣a' < 'a'↵
```

2.7. Módulos e importación de funciones y variables

Python también proporciona funciones trigonométricas, logaritmos, etc., pero no están directamente disponibles cuando iniciamos una sesión. Antes de utilizarlas hemos de indicar a Python que vamos a hacerlo. Para ello, *importamos* cada función de un módulo.

2.7.1. El módulo *math*

Empezaremos por importar la función seno (*sin*, del inglés «sinus») del módulo matemático (*math*):

```
>>> from math import sin↵
```

Ahora podemos utilizar la función en nuestros cálculos:

```
>>> from math import sin↵
>>> sin(0)↵
0.0
>>> sin(1)↵
0.8414709848078965
```

Observa que el argumento de la función seno debe expresarse en radianes.

Inicialmente Python no «sabe» calcular la función seno. Cuando importamos una función, Python «aprende» su definición y nos permite utilizarla. Las definiciones de funciones residen en *módulos*. Las funciones trigonométricas residen en el módulo matemático. Por ejemplo, la función coseno, en este momento, es desconocida para Python.

```
>>> cos(0)↵
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'cos' is not defined
```

Antes de usarla, es necesario importarla del módulo matemático:

```
>>> from math import cos↵
>>> cos(0)↵
1.0
```

En una misma sentencia podemos importar más de una función. Basta con separar sus nombres con comas:

```
>>> from math import sin, cos↵
```

Puede resultar tedioso importar un gran número de funciones y variables de un módulo. Python ofrece un atajo: si utilizamos un asterisco, se importan *todos* los elementos proporcionados por un módulo. Para importar todos los elementos del módulo *math* escribimos:

```
>>> from math import *↵
```

Así de fácil. De todos modos, no resulta muy aconsejable por dos razones:

- Al importar elemento a elemento, el programa gana en legibilidad, pues sabemos de dónde proviene cada identificador.
- Si hemos definido una variable con un nombre determinado y dicho nombre coincide con el de una función definida en un módulo, nuestra variable será sustituida por la función. Si no sabes todos los elementos que define un módulo, es posible que esta coincidencia de nombre tenga lugar, te pase inadvertida inicialmente y te lleves una sorpresa cuando intentes usar la variable.

He aquí un ejemplo del segundo de los problemas indicados:

```
>>> pow = 1↵
>>> from math import *↵
>>> pow += 1↵
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: unsupported operand type(s) for +=: 'builtin_function_or_method' and 'int'
```

Python se queja de que intentamos sumar un entero y una función. Efectivamente, hay una función *pow* en el módulo *math*. Al importar todo el contenido de *math*, nuestra variable ha sido «machacada» por la función.

Te presentamos algunas de las funciones que encontrarás en el módulo matemático:

$sin(x)$	Seno de x, que debe estar expresado en radianes.
$cos(x)$	Coseno de x, que debe estar expresado en radianes.
$tan(x)$	Tangente de x, que debe estar expresado en radianes.
$exp(x)$	El número e elevado a x.
$ceil(x)$	Redondeo hacia arriba de x (en inglés, «ceiling» significa techo).
$floor(x)$	Redondeo hacia abajo de x (en inglés, «floor» significa suelo).
$log(x)$	Logaritmo natural (en base e) de x.
$log10(x)$	Logaritmo decimal (en base 10) de x.
$sqrt(x)$	Raíz cuadrada de x (del inglés «square root»).

Evitando las coincidencias

Python ofrece un modo de evitar el problema de las coincidencias: indicar solo el nombre del módulo al importar.

```
>>> import math↵
```

De esta forma, todas las funciones del módulo *math* están disponibles, pero usando el nombre del módulo y un punto como prefijo:

```
>>> import math↵
>>> math.sin(0)↵
0.0
>>> math.cos(0)↵
1.0
```

En el módulo matemático se definen, además, algunas constantes de interés:

```
>>> from math import pi, e↵
>>> pi↵
3.141592653589793
>>> e↵
2.718281828459045
```

▶ 30 ¿Qué resultados se obtendrán al evaluar las siguientes expresiones Python? Calcula primero a mano el valor resultante de cada expresión y comprueba, con la ayuda del ordenador, si tu resultado es correcto.

a) `int(exp(2 * log(3)))`

b) `round(4*sin(3 * pi / 2))`

c) `abs(log10(.01) * sqrt(25))`

d) `round(3.21123 * log10(1000), 3)`

Precisión de los flotantes

Hemos dicho que los argumentos de las funciones trigonométricas deben expresarse en radianes. Como sabrás, $\text{sen}(\pi) = 0$. Veamos qué opina Python:

```
>>> from math import sin, pi↵
>>> sin(pi)↵
1.2246467991473532e-16
```

El resultado que proporciona Python no es cero, sino un número muy próximo a cero: 0.00000000000000012246467991473532. ¿Se ha equivocado Python? No exactamente. Ya dijimos antes que los números flotantes tienen una precisión limitada. El número π está definido en el módulo matemático como 3.141592653589793115997963468544185161590576171875, cuando en realidad posee un número infinito de decimales. Así pues, no hemos pedido exactamente el cálculo del seno de π, sino el de un número próximo, pero no exactamente igual. Por otra parte, el módulo matemático hace cálculos mediante algoritmos que pueden introducir errores en el resultado. Fíjate en el resultado de esta sencilla operación:

```
>>> 0.1 - 0.3↵
-0.19999999999999998
```

Los resultados con números en coma flotante deben tomarse como meras aproximaciones de los resultados reales.

2.7.2. Otros módulos de interés

Existe un gran número de módulos, cada uno de ellos especializado en un campo de aplicación determinado. Precisamente, una de las razones por las que Python es un lenguaje potente y extremadamente útil es por la gran colección de módulos con que se distribuye. Hay módulos para el diseño de aplicaciones para web, diseño de interfaces de usuario, compresión de datos, criptografía, multimedia, etc. Y constantemente aparecen nuevos módulos: cualquier programador de Python puede crear sus propios módulos, añadiendo así funciones que simplifican la programación en un ámbito cualquiera y poniéndolas a disposición de otros programadores. Nos limitaremos a presentarte ahora unas pocas funciones de un par de módulos interesantes.

Vamos con otro módulo importante: *sys* (sistema), el módulo de «sistema» (*sys* es una abreviatura del inglés «system»). Este módulo contiene funciones que acceden al sistema operativo y constantes dependientes del computador. Una función importante es *exit*, que aborta inmediatamente la ejecución del intérprete (en inglés significa «salir»). La variable *version*, indica con qué versión de Python estamos trabajando:

```
>>> from sys import version↵
>>> version↵
'3.2.3␣(default,␣Feb␣27␣2014,␣21:31:18)␣\n[GCC␣4.6.3]'
```

Y la variable *platform* permite saber sobre qué sistema operativo se está ejecutando el intérprete:

```
>>> from sys import platform↵
>>> platform↵
'linux2'
```

¡Ojo! Con esto no queremos decirte que las variables *version* o *platform* sean importantísimas y que debas aprender de memoria su nombre y cometido, sino que los módulos de Python contienen centenares de funciones y variables útiles para diferentes propósitos. Un buen programador Python sabe manejarse con los módulos. Existe un manual de referencia que describe todos los módulos estándar de Python. Lo encontrarás con la documentación Python bajo el nombre «*Library reference*» (en inglés significa «referencia de biblioteca») y podrás consultarla con un navegador web.

2.8. Métodos

Los datos de ciertos tipos permiten invocar unas funciones especiales: los denominados «*métodos*». Hemos visto que las funciones se invocan así: *función*(*argumento1*, *argumento2*, *argumento3*...). Los métodos son funciones especiales, pues se invocan del siguiente modo: *argumento1*.*método*(*argumento2*, *argumento3*...). Esta sintaxis recalca el hecho de que, para un método, el primer argumento es muy especial. Es como el sujeto de una frase de la que el método es el verbo.

2.8.1. Unos métodos sencillos para manipular cadenas...

Un método permite, por ejemplo, obtener una versión en minúsculas de la cadena sobre la que se invoca:

```
>>> cadena = 'Un␣EJEMPLO␣de␣Cadena'↵
>>> cadena.lower()↵
'un␣ejemplo␣de␣cadena'
>>> 'OTRO␣EJEMPLO'.lower()↵
'otro␣ejemplo'
```

La sintaxis es diferente de la propia de una llamada a función convencional. Lo primero que aparece es el propio objeto sobre el se efectúa la llamada. El nombre del método se separa del objeto con un punto. Los paréntesis abierto y cerrado al final son obligatorios.

Existe otro método, *upper* («uppercase», en inglés, significa «mayúsculas»), que pasa todos los caracteres a mayúsculas.

```
>>> 'Otro␣ejemplo'.upper()↵
'OTRO␣EJEMPLO'
```

Y otro, *title*, que pasa la inicial de cada palabra a mayúsculas. Te preguntarás para qué puede valer esta última función. Imagina que has hecho un programa de recogida de datos que confecciona un censo de personas y que cada individuo introduce personalmente su nombre en el ordenador. Es muy probable que algunos utilicen solo mayúsculas y otros mayúsculas y minúsculas. Si aplicamos *title* a cada uno de los nombres, todos acabarán en un formato único:

```
>>> 'PEDRO␣F.␣MAS'.title()↵
'Pedro␣F.␣Mas'
>>> 'Juan␣CANO'.title()↵
'Juan␣Cano'
```

Algunos métodos aceptan argumentos. El método *replace*, por ejemplo, recibe como argumento dos cadenas: un patrón y un reemplazo. El método busca el patrón en la cadena sobre la que se invoca el método y sustituye todas sus apariciones por la cadena de reemplazo.

```
>>> 'un␣pequeño␣ejemplo'.replace('pequeño', 'gran')↵
'un␣gran␣ejemplo'
>>> una_cadena = 'abc'.replace('b', '-')↵
>>> una_cadena↵
'a-c'
```

Complejos, decimales y fracciones

Python posee un rico conjunto de tipos de datos. Algunos, como los tipos de datos estructurados, se estudiarán con detalle más adelante. Sin embargo, y dado el carácter introductorio de este texto, no estudiaremos con detalle otros tipos de datos básicos: los números complejos, los números en formato decimal y las fracciones.

Un número complejo puro finaliza siempre con la letra jota, que representa el valor $\sqrt{-1}$. Un número complejo con parte real se expresa sumando la parte real a un complejo puro. He aquí ejemplos de números complejos: 4j, 1 + 2j, 2.0 + 3j, 1 - 0.354j. Y, para acabar, un ejemplo de expresiones aritméticas con complejos:

```
>>> (1 + 3j) / (2 + 1j)↵
(1+1j)
>>> (1 + 2j) * (1 - 2j)↵
(5+0j)
```

Los números de tipo decimal dan una solución al problema de la imprecisión de los flotantes. Esta imprecisión es inaceptable cuando manejamos, por ejemplo, dinero. El tipo *Decimal*, que hemos de importar del módulo *decimal*, maneja decimales con precisión:

```
>>> from decimal import Decimal↵
>>> a = Decimal('0.1')↵
>>> b = Decimal('0.3')↵
>>> a - b↵
Decimal('-0.2')
```

Otro modo de abordar el problema de la imprecisión de los flotantes es usar el tipo *Fraction*, que permite manipular números formados por un numerador y un denominador:

```
>>> from fractions import Fraction↵
>>> Fraction(2, 4)↵
Fraction(1, 2)
>>> Fraction(1, 10) - Fraction(3, 10)↵
Fraction(-1, 5)
```

Tanto *Decimal* como *Fraction* son sensiblemente más lentos que los flotantes. ¡No nos podía salir gratis!

2.8.2. ... y uno mucho más complejo: *format*

Aprender a mostrar la información con formato es esencial para que el usuario encuentre atractiva la forma en que ve los resultados. El método *format* de las cadenas nos proporciona una herramienta fundamental, aunque cuesta un poco dominarlo.

Empezamos por aprender a *interpolar* valores en una cadena, es decir, sustituir una marca especial por el valor de una expresión. Fíjate en esta sentencia:

```
>>> 'El␣número␣{0}␣ha␣sido␣interpolado.'.format(1.23)↵
'El␣número␣1.23␣ha␣sido␣interpolado.'
```

Los caracteres {0} han sido reemplazados por los caracteres 1.23. Ya sabíamos hacer esto de otro modo:

```
>>> 'El␣número␣{0}␣ha␣sido␣interpolado.'.replace('{0}', '1.23')↵
'El␣número␣1.23␣ha␣sido␣interpolado.'
```

El método *format* presenta algunas ventajas: su uso conduce a una expresión más breve y el número 1.23 se ha podido suministrar como número (cuando *replace* exige que sea una cadena). Pero las ventajas no acaban ahí. Podemos interporlar más de un valor con una sola llamada a *format*:

```
>>> 'Los␣números␣{0}␣y␣{1}␣han␣sido␣interpolados.'.format(1.23, 9.9999)↵
'Los␣números␣1.23␣y␣9.9999␣han␣sido␣interpolados.'
```

Cada marca de la forma {*n*}, donde *n* es un número entero, se sustituye por un argumento de *format*: la marca {0} se ha sustituido por el primer argumento y la marca {1} por el segundo. En general, la marca {*n*} se sustituye por el argumento $(n+1)$-ésimo. ¿No habría sido más sencillo para el diseñador de Python que {1} hiciese referencia al primer argumento, {2} al segundo y así sucesivamente? Ya te acostumbrarás a un principio básico de Python: *todas las secuencias empiezan en cero*. (Y no solo de Python: también C, Java y la mayoría de los lenguajes de uso común comparten ese principio).

Las marcas pueden disponerse dentro de la cadena en el orden que desees:

```
>>> 'Los␣números␣{1}␣y␣{0}␣han␣sido␣interpolados.'.format(1.23, 9.9999)↵
'Los␣números␣9.9999␣y␣1.23␣han␣sido␣interpolados.'
```

▶ 31 ¿Cuál será el resultado de evaluar estas expresiones?

```
>>> '{0}'.format(1)↵
>>> '{0}␣{1}'.format(1, 2)↵
>>> '{0}{1}'.format(1, 2)↵
>>> '{0},␣{1}'.format(1, 2)↵
>>> '{1},␣{0}'.format(1, 2)↵
>>> '{1},␣{1}'.format(1, 2)↵
>>> '{1},␣1'.format(1, 2)↵
```

Las marcas pueden modificarse para controlar el aspecto de la información. Imaginemos que deseamos mostrar los valores flotantes redondeados con un solo decimal:

```
>>> 'Los␣números␣{0:.1f}␣y␣{1:.1f}␣han␣sido␣interpolados.'.format(1.23, 9.9999)↵
'Los␣números␣1.2␣y␣10.0␣han␣sido␣interpolados.'
```

¿Complicado? Las marcas, a las que denominaremos en adelante *marcas de formato*, permiten controlar con precisión el modo en el que se muestran los datos. La forma general de una marca de formato es esta:

{*campo*!*marca de conversión*:*formato*}

El «campo» es el número que identifica el número de argumento que se desea interpolar en la cadena (aunque admite otros valores). Olvidemos por el momento el fragmento «!marca de conversión» y centrémonos en la parte «:formato». Su forma general es esta:

[[*relleno*]*alineamiento*][*signo*][#][0][*ancho*][.*precisión*][*código de tipo*]

Cada elemento entre corchetes es opcional. Es decir, el fragmento «signo» puede aparecer o no. Fíjate en que los corchetes se anidan en uno de los fragmentos. De acuerdo con ese anidamiento, una marca de formato puede empezar por un *relleno* o no hacerlo, pero solo puede tener un *relleno* si aparece también un *alineamiento*. Y ahora veamos algunas posibilidades para cada uno de esos elementos:

- *relleno*: Carácter con el que rellenar los espacios que requiere un alineamiento (por defecto, espacio en blanco).

- *alineamiento*: Carácter < para alinear a la izquierda en el espacio disponible; carácter > para alinear a la derecha en el espacio disponible (es el valor por defecto); carácter ^ para centrar en el espacio disponible.
- *signo*: Carácter + para forzar la aparición de un signo incluso en números positivos; carácter - para indicar que el signo solo debe aparecer con números negativos (es el valor por defecto); un espacio en blanco para indicar que los números positivos deben ir precedidos por un espacio en blanco.
- #: Si aparece, los enteros que se muestran en binario, octal o hexadecimal irán precedidos por `0b`, `0o` o `0x`.
- 0: Si aparece, se usa el carácter 0 para sustituir los espacios en blanco.
- *ancho*: Es un número entero que indica cuántos caracteres queremos que ocupe (como mínimo) el valor representado.
- `.`*precisión*: número de decimales con que queremos representar un número flotante.
- *código de tipo*: carácter que indica el tipo de representación que se desea. Es diferente según el tipo de datos del valor. He aquí algunos de sus posibles valores:
  - números enteros:
    - carácter `b`: en binario.
    - carácter `c`: como carácter Unicode.
    - carácter `d`: en base diez (es el valor por defecto).
    - carácter `o`: en octal.
    - carácter `x`: en hexadecimal.
    - carácter `n`: igual que `d`, pero como número adaptado a la cultura local (en español, por ejemplo, el punto decimal se muestra como una coma).
  - números flotantes:
    - carácter `e`: notación exponente.
    - carácter `f`: notación de punto fijo.
    - carácter `g`: notación en formato general, que solo es exponente para números grandes (es el valor por defecto).
    - carácter `n`: igual que `g`, pero como número adaptado a la cultura local.
    - carácter `%`: muestra el número multiplicado por 100, en formato `f` y seguido de un símbolo de porcentaje.

No has de memorizar esta lista de opciones y posibles valores, pero sí saber dónde está y recurrir a ella cuando la necesites. De todos modos, la mejor manera de ver qué hace exactamente cada opción es estudiar ejemplos de uso.

Empecemos con algunos enteros. Imprimamos el número 123 en su formato por defecto en medio de un texto:

```
>>> 'El {0} formateado.'.format(123)
'El 123 formateado.'
```

Ahora veamos cómo afectar de modos diferentes al alineamiento, siempre con una anchura de 10 espacios:

```
>>> 'El {0:>10} formateado.'.format(123)
'El        123 formateado.'
>>> 'El {0:^10} formateado.'.format(123)
'El    123    formateado.'
>>> 'El {0:<10} formateado.'.format(123)
'El 123        formateado.'
```

El 0 sustituye los espacios en blanco por un 0:

```
>>> 'El {0:010} formateado.'.format(123)
'El 0000000123 formateado.'
```

Veamos el efecto que consigue especificar un carácter de signo:

```
>>> 'El␣{0:+}␣formateado.'.format(123)↵
'El␣+123␣formateado.'
>>> 'El␣{0:-}␣formateado.'.format(123)↵
'El␣123␣formateado.'
>>> 'El␣{0:␣}␣formateado.'.format(123)↵
'El␣␣123␣formateado.'
```

El código de tipo permite mostrar el número en diferentes bases o incluso como carácter Unicode:

```
>>> 'El␣{0:b}␣formateado.'.format(123)↵
'El␣1111011␣formateado.'
>>> 'El␣{0:c}␣formateado.'.format(123)↵
'El␣{␣formateado.'
>>> 'El␣{0:d}␣formateado.'.format(123)↵
'El␣123␣formateado.'
>>> 'El␣{0:o}␣formateado.'.format(123)↵
'El␣173␣formateado.'
>>> 'El␣{0:x}␣formateado.'.format(123)↵
'El␣7b␣formateado.'
```

Con un # se muestra el prefijo que explicita la base cuando no es la decimal:

```
>>> 'El␣{0:#b}␣formateado.'.format(123)↵
'El␣0b1111011␣formateado.'
>>> 'El␣{0:#o}␣formateado.'.format(123)↵
'El␣0o173␣formateado.'
>>> 'El␣{0:#x}␣formateado.'.format(123)↵
'El␣0x7b␣formateado.'
```

Podemos combinar los diferentes elementos y controlar con mucha precisión el formato:

```
>>> 'El␣{0:+#016b}␣formateado.'.format(123)↵
'El␣+0b0000001111011␣formateado.'
```

No nos extenderemos tanto con las posibilidades de formato para números en coma flotante, pero no nos resistimos a poner algunos ejemplos que deberías analizar:

```
>>> 'El␣{0:e}␣formateado.'.format(123.45)↵
'El␣1.234500e+02␣formateado.'
>>> 'El␣{0:g}␣formateado.'.format(123.45)↵
'El␣123.45␣formateado.'
>>> 'El␣{0:+e}␣formateado.'.format(123.45)↵
'El␣+1.234500e+02␣formateado.'
>>> 'El␣{0:10e}␣formateado.'.format(123.45)↵
'El␣1.234500e+02␣formateado.'
>>> 'El␣{0:10.4g}␣formateado.'.format(123.45)↵
'El␣␣␣␣␣␣123.5␣formateado.'
>>> 'El␣{0:10.2g}␣formateado.'.format(123.45)↵
'El␣␣␣␣1.2e+02␣formateado.'
>>> 'El␣{0:10.1g}␣formateado.'.format(123.45)↵
'El␣␣␣␣␣␣1e+02␣formateado.'
>>> 'El␣{0:10.0g}␣formateado.'.format(123.45)↵
'El␣␣␣␣␣␣1e+02␣formateado.'
>>> 'El␣{0:.1%}␣formateado.'.format(123.45)↵
'El␣12345.0%␣formateado.'
```

Acabamos indicando que hay un pequeño problema: ¿Qué ocurre si queremos mostrar las llaves abierta o cerrada como tales en una cadena sometida a interpolación? Python se liará:

```
>>> 'Una␣{cadena}␣{0}'.format(1)↵
Traceback (most recent call last):
  File "<input>", line 1, in <module>
KeyError: 'cadena'
```

Las llaves que queramos mostrar como tales y que no sean objeto de sustitución al interpolar deben marcarse con dos apariciones seguidas de cada una de ellas:

```
>>> 'Una␣{{cadena}}␣{0}'.format(1)↵
'Una␣{cadena}␣1'
```

Como ves, hemos entrado en un tema lleno de detalles. Afortunadamente no usaremos muchos de los elementos que acabamos de presentar. Y si alguna vez te hicieran falta, siempre podrás consultar los manuales de ayuda.

Capítulo 3

Programas

> —¡Querida, realmente *tengo* que conseguir un lápiz más fino! No puedo en absoluto manejar este: escribe todo tipo de cosas, sin que yo se las dicte.
>
> *Alicia en el país de las maravillas*, Lewis Carroll

Hasta el momento hemos utilizado Python en un entorno interactivo: hemos introducido expresiones (y asignaciones a variables) y Python las ha evaluado y ha proporcionado inmediatamente sus respectivos resultados.

Pero utilizar el sistema únicamente de este modo limita bastante nuestra capacidad de trabajo. En este capítulo aprenderemos a introducir secuencias de expresiones y asignaciones en un fichero de texto y pedir a Python que las ejecute todas, una tras otra. Denominaremos *programa* al contenido del fichero de texto[1].

Puedes crear los ficheros con cualquier editor de texto. Nosotros utilizaremos un *entorno integrado de desarrollo* o IDE (por el inglés «Integrated Development Environment»): el entorno Eclipse con la extensión Pydev. Un entorno de programación es un conjunto de herramientas que facilitan el trabajo del programador. Eclipse es un IDE gratuito diseñado inicialmente para desarrollar programas con Java. Ocurre que Eclipse es un entorno extensible, es decir, se puede añadir funcionalidad y hacerlo útil para cometidos distintos del original. Pydev es una extensión de Eclipse para desarrollar programas con Python. Instalar y configurar Eclipse con Pydev apropiadamente no es una tarea trivial.

En este capítulo asumimos que dispones de un entorno Eclipse con Pydev correctamente configurado. En principio, cada programa autónomo debería tener su propio proyecto, pero nosotros crearemos ahora un único proyecto en el que iremos creando programas como módulos del mismo.

3.1. Tu primer programa

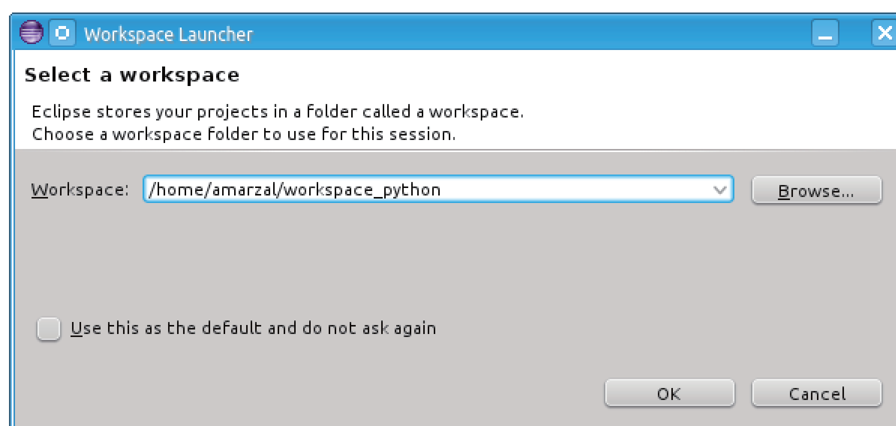
3.1.1. Instalar y preparar Eclipse para el trabajo con la extensión Pydev

Arranca Eclipse. Aparece un pantallazo de presentación similar al siguiente (que es el de la versión 4.3 de Eclipse, conocida como Kepler):

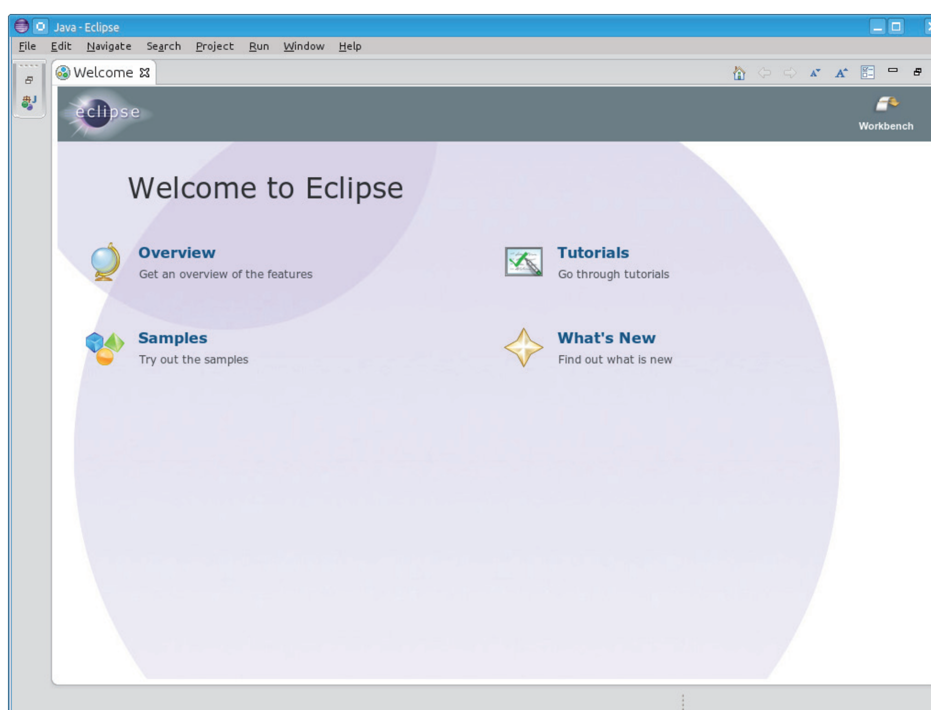


[1] También se suele denominar *scripts* a los programas Python.

A continuación aparecerá un cuadro de diálogo en el que se te pedirá que indiques el espacio de trabajo («workspace») en el que vas a moverte en esta sesión. El espacio de trabajo es una carpeta (un directorio) en el que puedes agrupar proyectos relacionados entre sí. Es recomendable que uses un único espacio de trabajo para el trabajo con este texto. Cada programa con suficiente entidad, cuando los haya, podrá crearse en su propio proyecto dentro del espacio de trabajo. Como es la primera vez que trabajamos con Eclipse, vamos a crear un espacio de trabajo propio al que denominaremos `workspace_python`. Modifica el nombre que te propone por defecto para que quede así[2] y pulsa el botón OK:

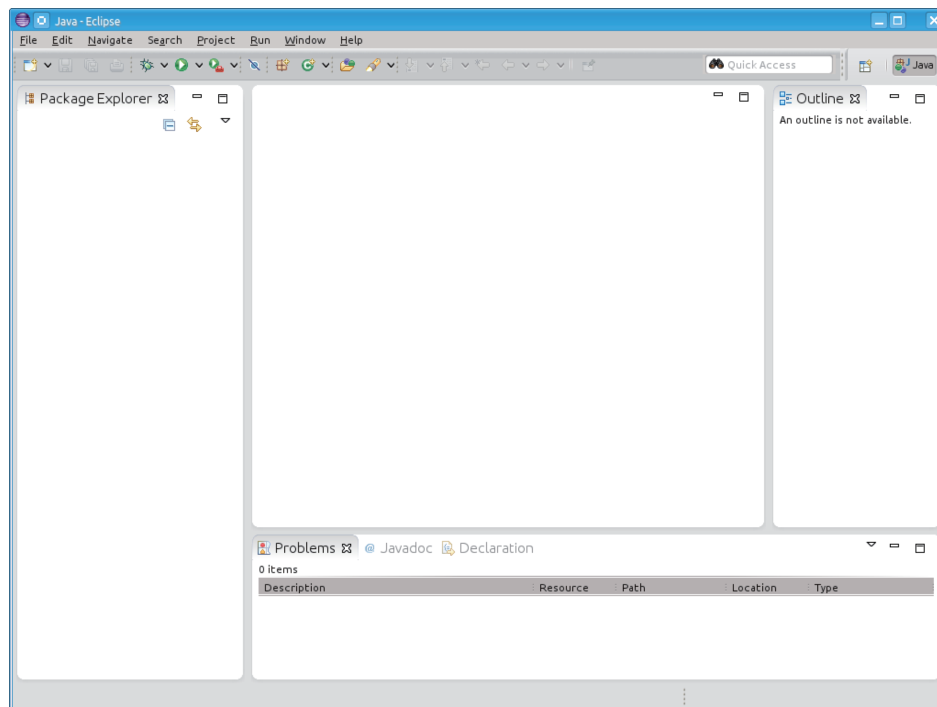


Probablemente Eclipse arranque con una pantalla como esta:

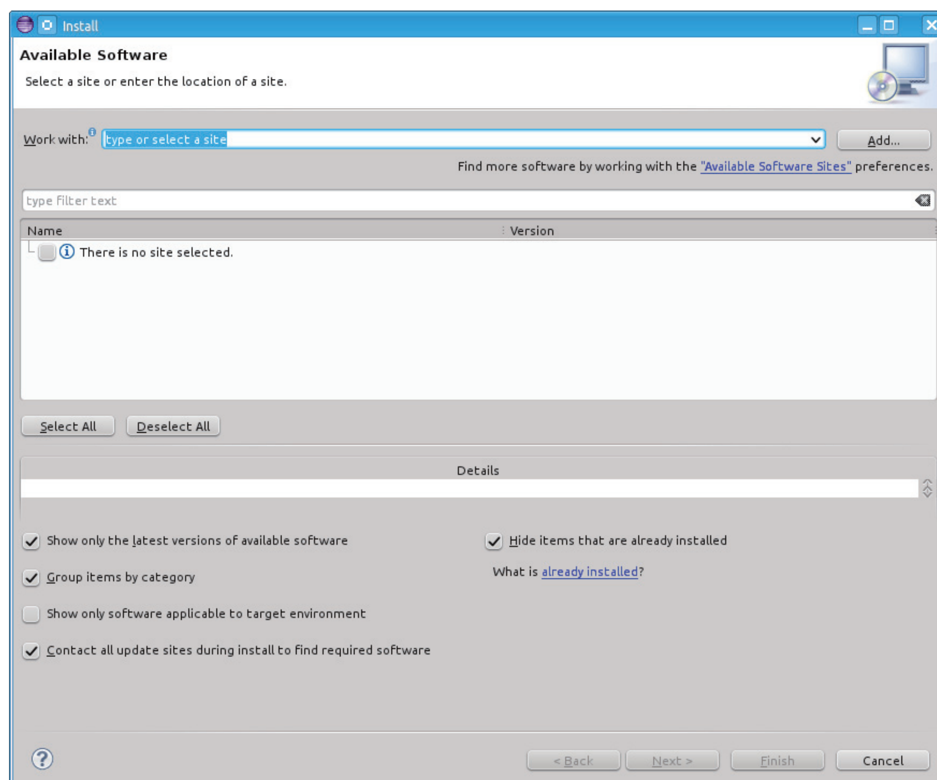


Es una pantalla de bienvenida que conduce a una serie de manuales y tutoriales. Como estos están principalmente orientados al trabajo con Java, no nos resultan de mucha utilidad ahora mismo. Lo mejor es que cerremos la pestaña titulada «Welcome» pulsando en el aspa que hay a su derecha. Encontrarás entonces un entorno de trabajo como este:

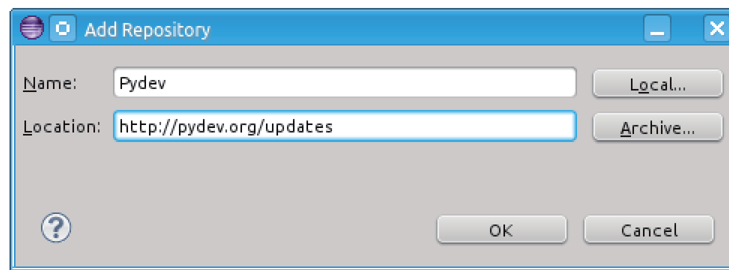
[2]La ruta del espacio de trabajo será diferente en función del sistema operativo con el que estés trabajando. Las imágenes han sido capturadas en un ordenador con sistema operativo Linux.



Hemos de instalar ahora el módulo Pydev, que adapta Eclipse al trabajo con Python. Necesitarás una conexión a Internet y haber instalado previamente Python 3.1 (o superior). Ve al menú *Help→Install New Software*. Aparecerá un cuadro de diálogo como este:

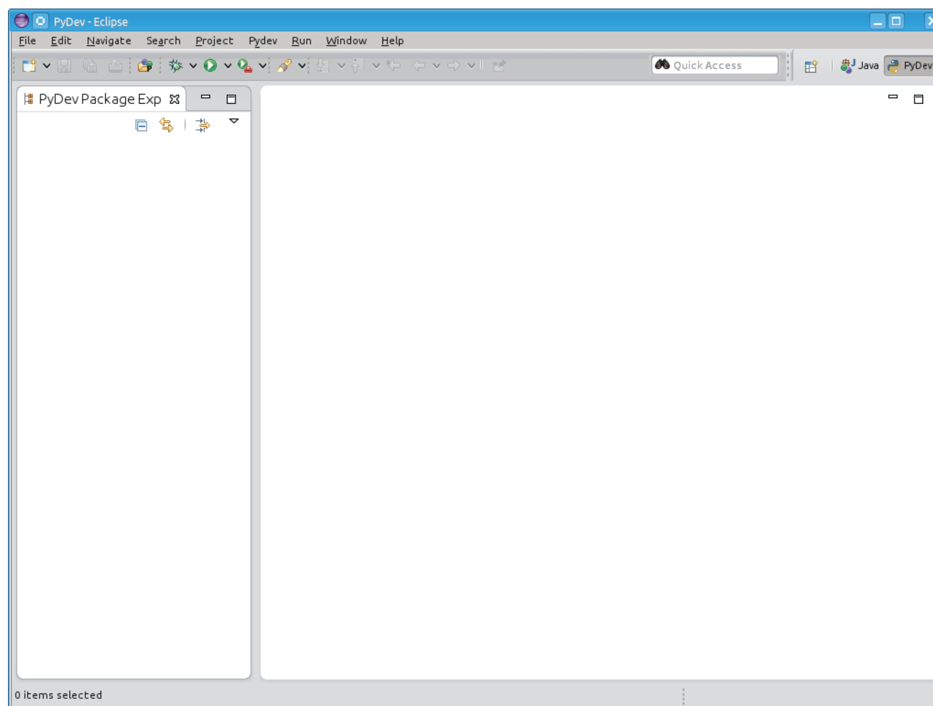


Pulsa el botón «Add». Aparecerá un nuevo cuadro de diálogo. En el campo «Name:» escribe el texto `Pydev` y en el campo «Location» escribe la dirección `http://pydev.org/updates`:



Pulsa ahora el botón «Ok». En la nueva pantalla que aparece, marca la casilla «PyDev» y avanza dos pantallas más pulsando «Next». A continuación, acepta las condiciones que impone una licencia de uso, pulsa «Finish» e indica que confías en el certificado del sitio de instalación para que Eclipse proceda a instalar el componente Pydev. Por último, será necesario reiniciar Eclipse para completar el proceso de instalación.

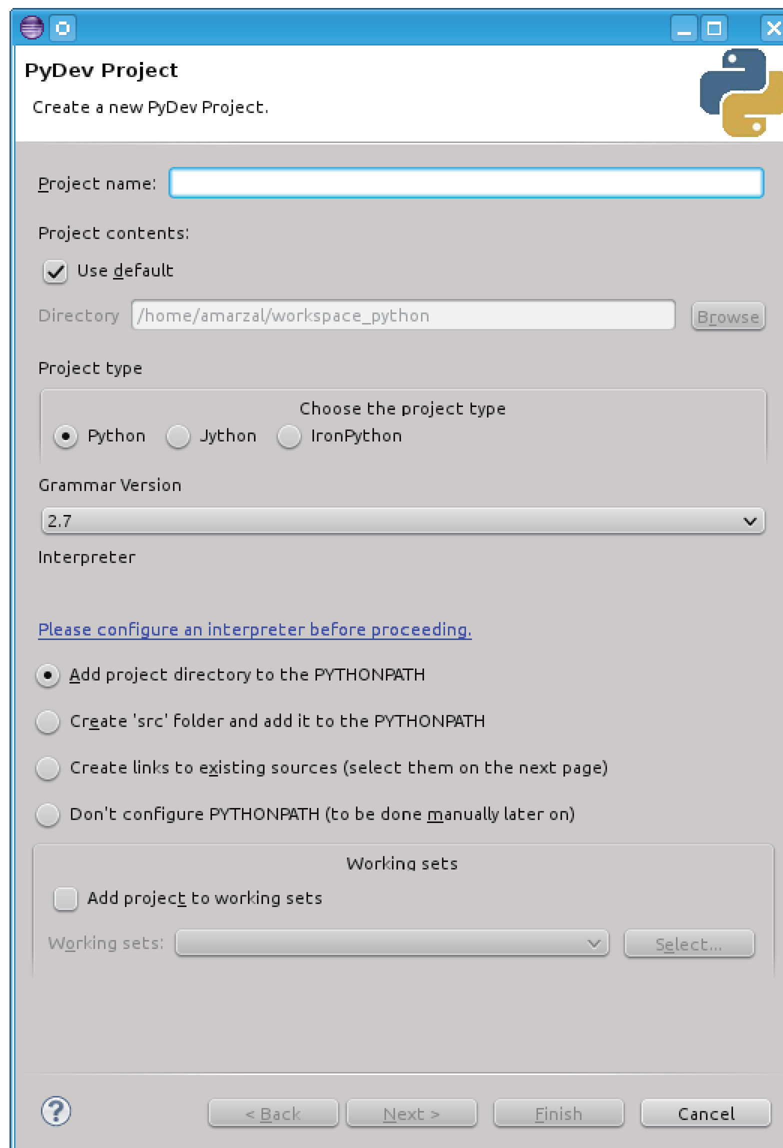
Solo nos queda un paso antes de crear un proyecto Pydev: seleccionar la perspectiva Pydev. Una perspectiva Eclipse es un conjunto de paneles dispuestos de una determinada forma que facilitan el desarrollo con un determinado lenguaje, tipo de ficheros o plataforma. Para elegir la perspectiva Pydev ve al menú *Window→Open Perspective→Other*, selecciona *Pydev* en el cuadro de diálogo que se habrá abierto y pulsa «OK». La apariencia de la ventana principal cambiará un poco:



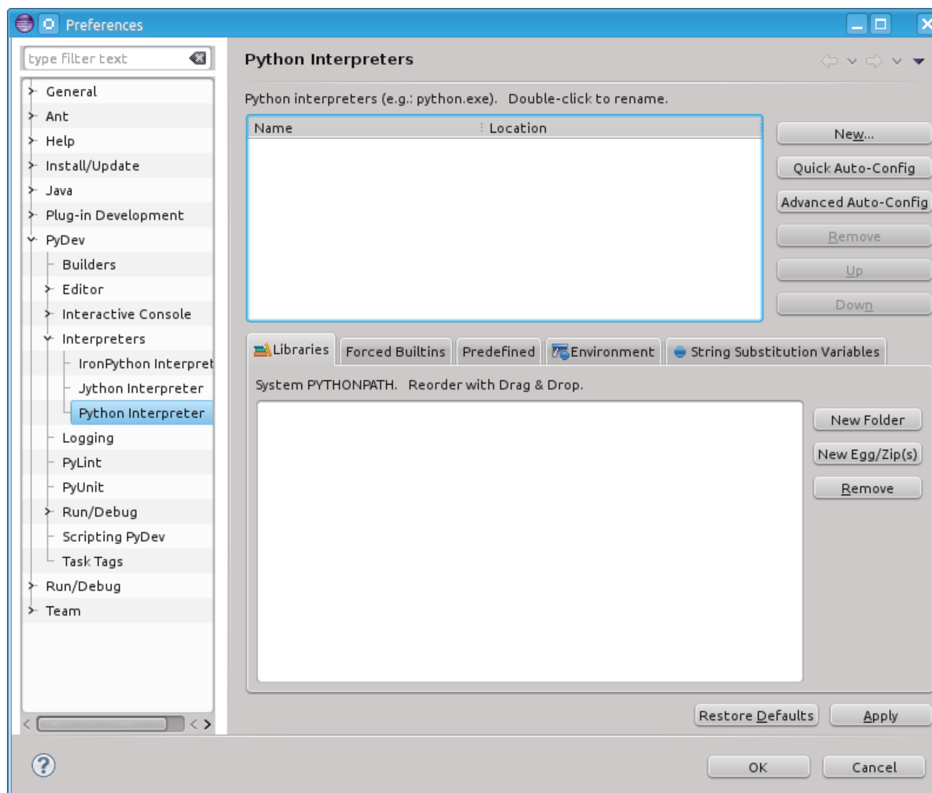
La ventana principal presenta dos regiones o vistas[3] (del inglés «views»). En la izquierda encontrarás el «Pydev Package Explorer» y en él aparecerán los diferentes proyectos que crees y, dentro de cada proyecto, las carpetas y ficheros que lo formen. A su derecha hay un espacio para albergar los editores de texto con los que crearás y modificarás los programas.

Empecemos creando un proyecto al que llamaremos `primeros_programas`. Selecciona la opción de menú *File→New→Pydev Project*. Aparecerá un cuadro de diálogo como este:

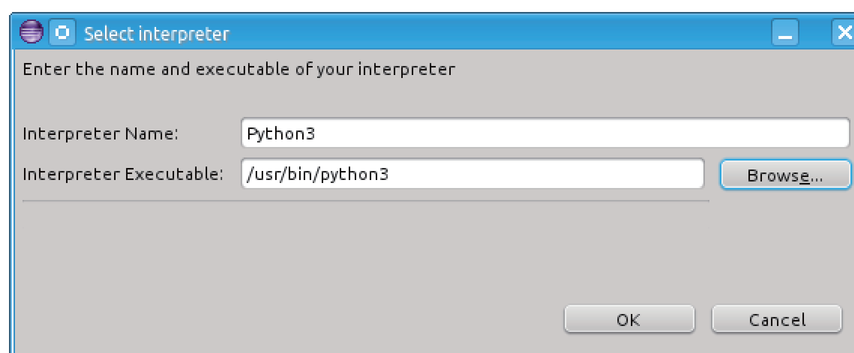
[3] Hay más vistas, que puedes activar mediante *Window→Show View*. Por ejemplo, la vista «Outline» permite mostrar una visión esquemática del programa cuyo editor tenga el foco y te resultará muy útil para navegar rápidamente por tus programas cuando estos tengan decenas (o centenares) de líneas de código.



El nombre del proyecto (campo «Project name») será `primeros_programas`. Dejaremos marcada la opción «Use default» para que Pydev cree la estructura del proyecto. El tipo de proyecto es «Python» y la versión de la gramática es 3.0 (aunque usemos Python 3.1). Y como este es nuestro primer proyecto, hemos de hacer un trabajo extra: dar de alta el intérprete Python que usaremos para ejecutar nuestro programa en el entorno. Para ello pinchamos en el enlace «Please configure an interpreter before proceeding» y seleccionamos «Manual Config». Esto nos llevará a un nuevo cuadro de diálogo:



Pulsa en el botón «New» que hay arriba a la derecha. Aparecerá un nuevo cuadro de diálogo titulado «Select interpreter». El primer campo debe contener «Python3»[4]. En un sistema Unix, como Linux, el segundo campo deberá contener una ruta similar a `/usr/bin/python3`. Si estás trabajando con Microsoft Windows, el segundo campo contendrá `C:\Python31\python.exe`[5]:

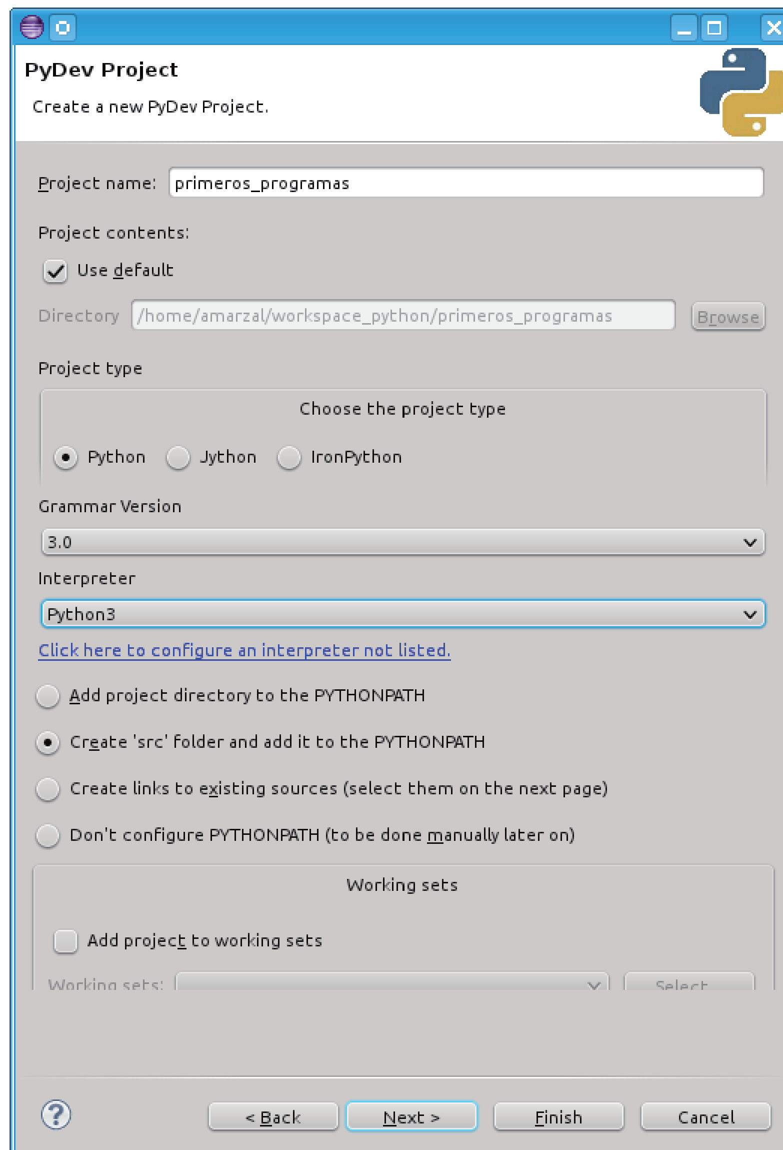


Seguimos. Pulsamos «OK» y aparece un nuevo cuadro de diálogo con una relación de rutas. Pulsamos nuevamente «OK». Regresamos así al cuadro de diálogo «Preferences» y en él pulsamos ahora «OK». Pydev se tomará su tiempo para configurar el sistema. (Recuerda que esto tan trabajoso solo lo hacemos una vez por espacio de trabajo, así que la pérdida de tiempo no se repetirá mucho). En el cuadro de diálogo aparece, bajo el desplegable para seleccionar la gramática, un nuevo desplegable para seleccionar un intérprete. Podemos dejarlo tal cual está, con la opción «Default», o seleccionar la opción «Python3» (pues en este momento solo hemos dado de alta un intérprete y es lo mismo seleccionar «Default» que «Python3», es decir, la etiqueta con la que hemos nombrado dicho intérprete). Es habitual que almacenemos nuestros

[4] Lo cierto es que da igual el texto del primer campo: no es más que una etiqueta. Eso sí: usemos un nombre sencillo que deje claro que usamos un intérprete Python de la versión 3.

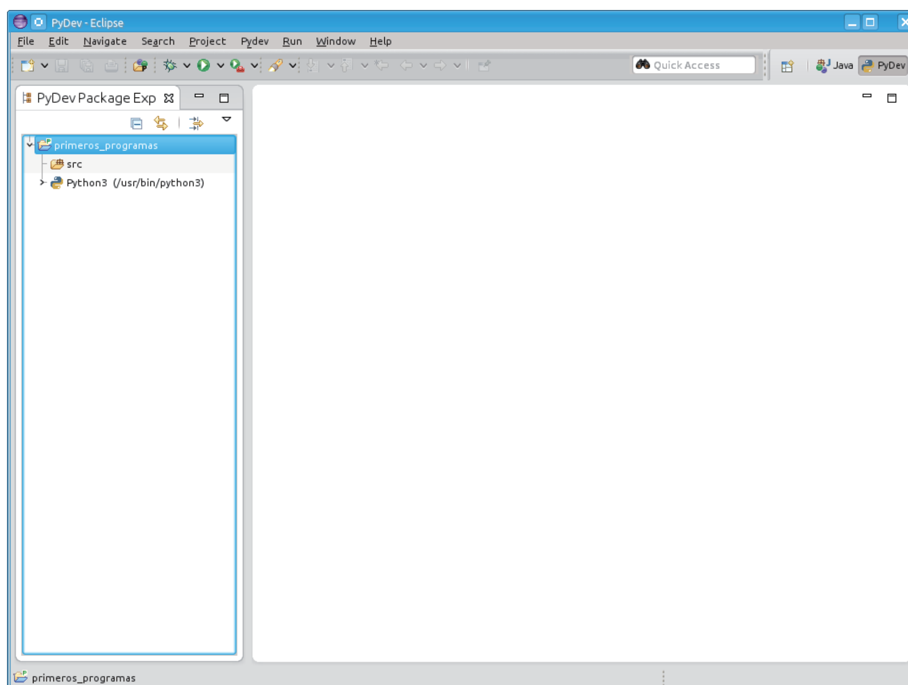
[5] Tanto en Unix como en Windows puede haber alguna diferencia con la ruta del intérprete. Todo depende del modo en que hiciste la instalación del paquete Python 3. Las rutas que indicamos corresponden a instalaciones estándar de Python 3.1.

programas en una carpeta llamada «src» (del inglés «source»). Para ello, marcaremos la opción «Create 'src' folder»:

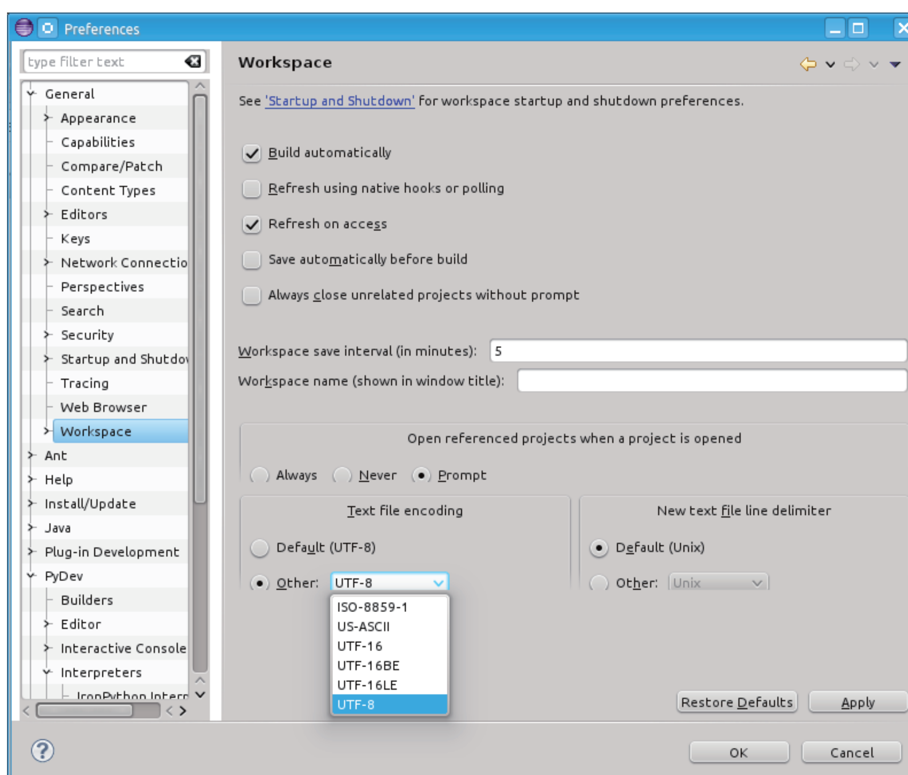


Pulsamos «Finish» y ya estamos listos para trabajar.

Fíjate en que en la vista del explorador de paquetes Python aparece una carpeta con el nombre `primeros_programas`. Si acercamos el ratón a ella comprobaremos que se trata de una carpeta desplegable. Al pulsar en el botón triangular que hay a su izquierda, se desplegará la carpeta y se mostrará su contenido:



Nos queda aún un pequeño detalle para tenerlo todo listo. Nos gustaría fijar una codificación de texto para todos los ficheros que más tarde no nos dé problemas con la representación de los caracteres acentuados o de la letra eñe. Podemos hacer esto fichero a fichero, pero es mucho mejor fijarlo en las preferencias del propio espacio de trabajo. Has de ir al menú *Window→Preferences*. Aparecerá un cuadro de diálogo complejo. Selecciona la opción *General→Workspace* en el árbol de menús que hay a mano izquierda. En el panel de la derecha aparecerá un cuadro de diálogo con un elemento titulado «Text file encoding». Ese elemento contiene dos botones de radio con las opciones «Default» y «Other». Si es necesario, selecciona la opción «Other» y, en el menú desplegable que se activa entonces, selecciona «UTF-8». Cierra finalmente el cuadro pulsando el botón «OK»:



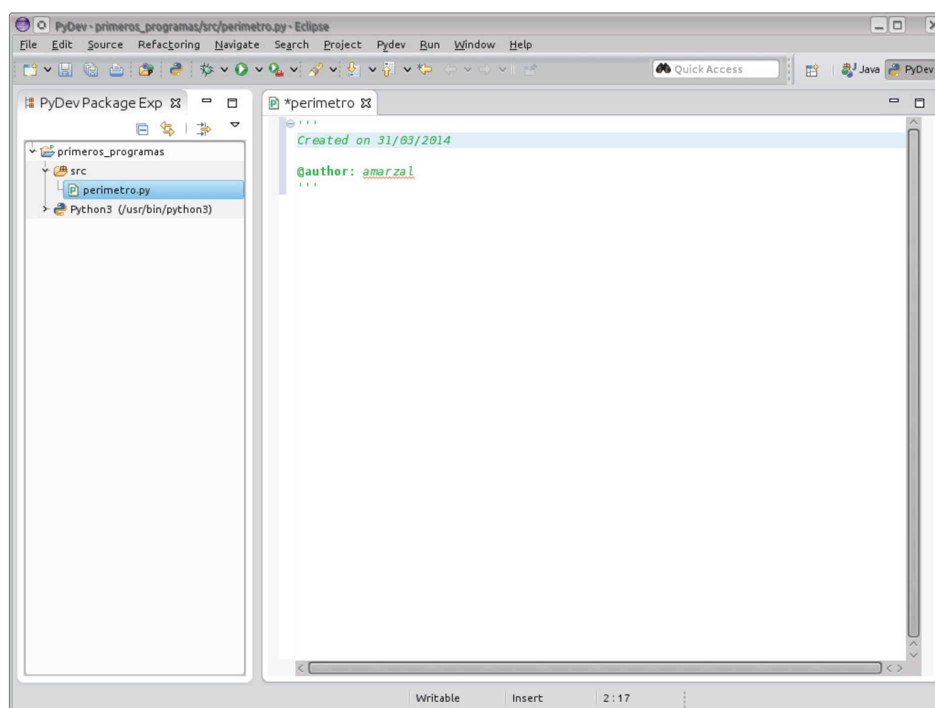
Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

Con eso hemos seleccionado el juego de caracteres Unicode y nuestros ficheros podrán leerse igual en cualquier sistema moderno. Ya lo tenemos todo preparado para empezar a escribir nuestro primer programa.

3.1.2. Nuestro primer programa

Empezaremos por crear el fichero en el que escribiremos el programa. Pulsa el botón derecho en la carpeta «src» y en el menú contextual selecciona *New→Pydev module*. En el cuadro de diálogo no edites el primer campo, que dice `/primeros_programas/src`, ni el segundo, que está en blanco. En el tercero escribe `perimetro`[6] y pulsa «OK». En el nuevo cuadro de diálogo que aparece, deja la selección de «Template» en la opción «Empty», tal cual está. (Seguramente estás ya abrumado por lo trabajoso que es configurar el entorno para crear un proyecto y trabajar en él. Tranquilo. Acabará siendo un proceso mecánico).

Acabamos de crear el fichero `perimetro.py` y se ha abierto un editor de texto para que podamos modificar su contenido. El fichero no está en blanco: contiene un texto que proviene de una plantilla. El texto contiene la fecha de creación y el nombre del autor:



Empezaremos por eliminar ese texto (ya veremos más adelante para qué podría servir) y escribir en su lugar este otro:

perimetro.py
```python
from math import pi
radio = 1
perímetro = 2 * pi * radio
perímetro
```

Si hubiésemos escrito cada una de estas líneas directamente en el intérprete interactivo de Python, como hacíamos en el capítulo anterior, la ejecución de la última línea mostraría el resultado en pantalla. Vamos a ejecutar todas las líneas del programa con una sola orden del entorno de desarrollo: selecciona la opción de menú *Run→Run* y elige el modo «Python Run».

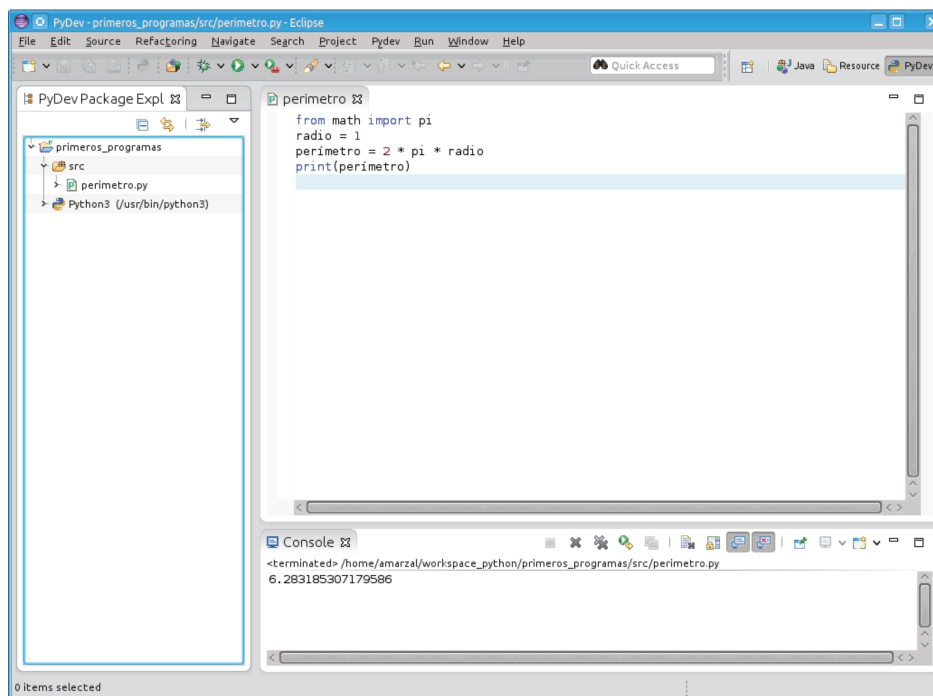
[6] Ya habrás detectado que la palabra `perimetro` está mal escrita: le falta la tilde a la i. Dado que la codificación de los nombres de fichero es una cuestión delicada, evitaremos el uso de caracteres que no forman parte de la tabla ASCII, aunque ello suponga tener que cometer faltas de ortografía. Puede que esto te quite el sueño, pero de momento no tenemos otro remedio para evitar problemas de portabilidad de los ficheros.

> **Punto py**
>
> Hay un convenio por el que los ficheros que contienen programas Python tienen extensión en su nombre. La extensión de un nombre de fichero son los caracteres del mismo que suceden al (último) punto. Un fichero llamado **`ejemplo.py`** tiene extensión **`py`**.
>
> La idea de las extensiones viene de antiguo y es un mero convenio. Puedes prescindir de él, pero no es conveniente. En entornos gráficos (como KDE, Gnome, Max OS X o Microsoft Windows) la extensión se utiliza para determinar qué icono va asociado al fichero y qué aplicación debe arrancarse para abrir el fichero al hacer clic (o doble clic) en el mismo.

¡No ocurre nada! Python se comporta de modo diferente en función de si se usa interactivamente o ejecutando un programa escrito en un fichero. En el primer caso, cada expresión (y la última línea del programa es una expresión) provoca que se muestre por pantalla el resultado de su evaluación. En el segundo, es necesario decir explícitamente al intérprete que deseamos imprimir un valor por pantalla. Para eso tenemos la función *print*:

perimetro.py
```
from math import pi
radio = 1
perímetro = 2 * pi * radio
print(perímetro)
```

Si ahora lo ejecutamos nuevamente, en la zona inferior aparecerá una nueva vista con la consola y, en ella, la salida del programa:



Una última observación. Los programas deben ser legibles y conviene que juguemos con un elemento muy básico: las líneas en blanco. Es buen estilo separar las diferentes zonas del programa con líneas en blanco. En nuestro programa hay tres regiones: la importación de elementos de la librería matemática, los cálculos y la presentación del resultado. El programa resulta más legible formateado así:

perimetro.py
```
from math import pi

radio = 1
```

```
perímetro = 2 * pi * radio

print(perímetro)
```

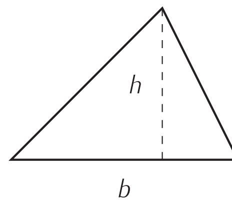
Puede que te parezca una tontería, pero cuando tengas un programa de cientos de líneas con algún error y hayas de leerlo una y otra vez hasta detectar ese error, agradecerás todo esfuerzo puesto en aumentar la legibilidad del programa.

▶ 32 Diseña un programa que, a partir del valor del lado de un cuadrado (3 metros), muestre el valor de su perímetro (en metros) y el de su área (en metros cuadrados).

(El perímetro debe darte 12 metros y el área 9 metros cuadrados).

▶ 33 Diseña un programa que, a partir del valor de la base y de la altura de un triángulo (3 y 5 metros, respectivamente), muestre el valor de su área (en metros cuadrados).

Recuerda que el área A de un triángulo se puede calcular a partir de la base b y la altura h como $A = \frac{1}{2}bh$.



(El resultado es 7.5 metros cuadrados).

▶ 34 Diseña un programa que, a partir del valor de los dos lados de un rectángulo (4 y 6 metros, respectivamente), muestre el valor de su perímetro (en metros) y el de su área (en metros cuadrados).

(El perímetro debe darte 20 metros y el área 24 metros cuadrados).

3.2. Ejecución de programas desde la línea de órdenes

El programa que hemos escrito no solo puede ejecutarse desde el entorno de desarrollo. Abre un intérprete de órdenes (en Unix, cualquier terminal te vale; en Windows escribe **cmd** en la caja de búsqueda del icono de inicio, situado en la esquina inferior izquierda de la pantalla, y pulsa retorno) y ve al directorio en el que se encuentra el fichero **perimetro.py**.

En Unix escribe **cd workspace_python/primeros_programas/src** y pulsa retorno de carro; en Windows escribe **cd workspace_python\primeros_programas\src** y pulsa retorno de carro. A continuación, escribe **python3 perimetro.py**. El sistema responderá imprimiendo en pantalla el resultado de ejecutar el programa.

3.3. Entrada/salida

Los programas que hemos visto en la sección anterior adolecen de un serio inconveniente: cada vez que quieras obtener resultados para unos datos diferentes deberás editar el fichero de texto que contiene el programa.

Por ejemplo, el siguiente programa calcula el volumen de una esfera a partir de su radio, que es de un metro:

esfera.py
```
from math import pi

radio = 1
volumen = 4 / 3 * pi * radio ** 3
```

```
5
6 print(volumen)
```

El programa se ha escrito en un nuevo fichero del proyecto **`primeros_programas`**. El fichero, llamado **`esfera.py`** se ha creado pulsando el botón derecho del ratón sobre la carpeta **`src`** del proyecto y seleccionando *New→Pydev module*. En el cuadro de diálogo hemos escrito **`esfera`** en el campo **`Name`** y hemos pulsado el botón «Finish».

Al ejecutar el programa obtenemos en pantalla este texto:

```
4.1887902047863905
```

Si deseas calcular ahora el volumen de una esfera de 3 metros de radio, debes editar el fichero que contiene el programa, yendo a la tercera línea y cambiándola para que el programa pase a ser este:

esfera.py
```
1 from math import pi
2
3 radio = 3
4 volumen = 4 / 3 * pi * radio ** 3
5
6 print(volumen)
```

Al ejecutar nuevamente el programa obtenemos en pantalla este otro texto:

```
113.09733552923254
```

Y si ahora quieres calcular el volumen para otro radio, vuelta a empezar: ve a la tercera línea, modifica el valor del radio y ejecuta. No es el colmo de la comodidad.

3.3.1. Lectura de datos de teclado

Vamos a aprender a hacer que nuestro programa, cuando se ejecute, pida el valor del radio para el que vamos a efectuar los cálculos *sin necesidad de editar el fichero de programa*.

Hay una función predefinida, *input* (en inglés significa «entrada»), que hace lo siguiente: detiene la ejecución del programa y espera a que el usuario escriba un texto (el valor del radio, por ejemplo) y pulse la tecla de retorno de carro; en ese momento prosigue la ejecución y la función devuelve *una cadena* con el texto que tecleó el usuario.

Si deseas que el radio sea un valor flotante, debes transformar la cadena devuelta por *input* en un dato de tipo flotante llamando a la función *float*. La función *float* recibirá como argumento la cadena que devuelve *input* y proporcionará un número en coma flotante. (Recuerda, para cuando lo necesites, que existe otra función de conversión, *int*, que devuelve un entero en lugar de un flotante). Por otra parte, *input* es una función y, por tanto, el uso de los paréntesis que siguen a su nombre es obligatorio, incluso cuando no tenga argumentos.

Modificamos el fichero **`esfera.py`** anterior para que quede de la siguiente forma:

esfera.py
```
1 from math import pi
2
3 cadena_leída = input()
4 radio = float(cadena_leída)
5
6 volumen = 4 / 3 * pi * radio ** 3
7
8 print(volumen)
```

Ejecuta ahora el programa. Si estás en el entorno de desarrollo Eclipse/Pydev, pincha en la vista de la consola y teclea el valor del radio. Escribe, por ejemplo, el valor 2. Pulsa a continuación el retorno de carro y en la misma vista de consola aparecerá el volumen. Este es el resultado de la ejecución, donde el texto que teclea el usuario aparece en un color distinto y

va seguido de la marca de retorno de carro (para recordarte que debes pulsarlo tras introducir el dato):

```
2↲
33.510321638291124
```

El programa no es muy elegante por el modo en que pide el dato de entrada: la consola se queda bloqueada y no sale ningún mensaje que alerte al usuario de que se le pide un dato en particular. La función *input* admite un argumento opcional: una cadena con el texto que debe mostrarse en pantalla para que el usuario sepa qué introducir. Esta otra versión del programa es más elegante:

esfera.py
```
from math import pi

cadena_leída = input('Dame␣el␣radio:␣')
radio = float(cadena_leída)

volumen = 4 / 3 * pi * radio ** 3

print(volumen)
```

El usuario verá ahora un mensaje de texto «Dame el radio: » e introducirá el valor del radio como respuesta a esta petición. Volvamos a ejecutar el programa introduciendo el valor 2 como radio.

```
Dame el radio: 2↲
33.510321638291124
```

¡Mucho mejor!

Para acabar, ten en cuenta que es posible juntar en una sola las dos líneas dedicadas a la lectura del dato:

esfera.py
```
from math import pi

radio = float(input('Dame␣el␣radio:␣'))

volumen = 4 / 3 * pi * radio ** 3

print(volumen)
```

▶ 35 Diseña un programa que pida el valor del lado de un cuadrado y muestre el valor de su perímetro y el de su área. (Prueba que tu programa funciona correctamente con este ejemplo: si el lado vale 1.1, el perímetro será 4.4, y el área 1.21).

▶ 36 Diseña un programa que pida el valor de los dos lados de un rectángulo y muestre el valor de su perímetro y el de su área. (Prueba que tu programa funciona correctamente con este ejemplo: si un lado mide 1 y el otro 5, el perímetro será 12.0, y el área 5.0).

▶ 37 Diseña un programa que pida el valor de la base y la altura de un triángulo y muestre el valor de su área. (Prueba que tu programa funciona correctamente con este ejemplo: si la base es 10 y la altura 100, el área será 500.0).

▶ 38 Diseña un programa que pida el valor de los tres lados de un triángulo y calcule el valor de su área y perímetro. Recuerda que el área A de un triángulo puede calcularse a partir de sus tres lados, a, b y c, así: $A = \sqrt{s(s-a)(s-b)(s-c)}$, donde $s = (a+b+c)/2$. (Prueba que tu programa funciona correctamente con este ejemplo: si los lados miden 3, 5 y 7, el perímetro será 15.0 y el área 6.49519052838).

3.3.2. Más sobre la función *print*

Las cadenas pueden usarse también para mostrar textos por pantalla en cualquier momento a través de sentencias *print*.

```
esfera.py
from math import pi

print('Programa␣para␣el␣cálculo␣del␣volumen␣de␣una␣esfera.')

radio = float(input('Dame␣el␣radio␣(en␣metros):␣'))

volumen = 4 / 3 * pi * radio ** 3

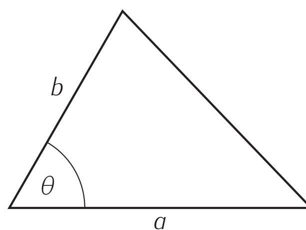
print('Volumen:', volumen, 'metros␣cúbicos.')
print('Gracias␣por␣usar␣el␣programa.␣Adiós.')
```

La primera aparición de *print* muestra en pantalla un mensaje que informa al usuario del propósito del programa. La segunda aparición de *print* muestra dos cosas en pantalla: el texto «Volumen:», el valor del volumen de la esfera y las unidades en las que se expresa el volumen. La función *print* puede mostrar en una misma línea más de un valor: los valores que se desee mostrar van entre paréntesis y separados por comas. Finalmente, la última aparición de *print* hace que se muestre un texto de agradecimiento y despedida.

Cuando ejecutes este programa, fíjate en que las cadenas que se muestran con *print* no aparecen entrecomilladas. El usuario del programa no está interesado en saber que le estamos mostrando datos del tipo cadena: solo le interesa el texto de dichas cadenas. Mucho mejor, pues, no mostrarle las comillas:

```
Programa para el cálculo del volumen de una esfera.
Dame el radio (en metros): 2↵
Volumen: 33.510321638291124 metros cúbicos.
Gracias por usar el programa. Adiós.
```

▶ 39 El área A de un triángulo se puede calcular a partir del valor de dos de sus lados, a y b, y del ángulo θ que estos forman entre sí con la fórmula $A = \frac{1}{2}ab\sin(\theta)$. Diseña un programa que pida al usuario el valor de los dos lados (en metros), el ángulo que estos forman (en grados), y muestre el valor del área.



(Ten en cuenta que la función *sin* de Python trabaja en radianes, así que el ángulo que leas en grados deberás pasarlo a radianes sabiendo que π radianes son 180 grados. Prueba que has hecho bien el programa introduciendo los siguientes datos: $a = 1$, $b = 2$, $\theta = 30$; el resultado es 0.5).

▶ 40 Haz un programa que pida al usuario una cantidad de euros, una tasa de interés y un número de años. Muestra por pantalla en cuánto se habrá convertido el capital inicial transcurridos esos años si cada año se aplica la tasa de interés introducida.

Recuerda que un capital de C euros a un interés del x por cien durante n años se convierten en $C \cdot (1 + x/100)^n$ euros. (Prueba tu programa sabiendo que una cantidad de 10,000 € al 4.5 % de interés anual se convierte en 24,117.14 € al cabo de 20 años).

▶ 41 Haz un programa que pida el nombre de una persona y lo muestre en pantalla repetido 1000 veces, pero dejando un espacio de separación entre aparición y aparición del nombre. (Utiliza los operadores de concatenación y repetición).

Por lo visto hasta el momento, cada *print* empieza a imprimir en una nueva línea. Esto es porque cada *print* añade al final un carácter especial: un terminador de línea. Podemos evitarlo si indicamos que no hay terminador de línea. Fíjate en este programa:

esfera.py
```
from math import pi

print('Programa␣para␣el␣cálculo␣del␣volumen␣de␣una␣esfera.')

radio = float(input('Dame␣el␣radio␣(en␣metros):␣'))

volumen = 4 / 3 * pi * radio ** 3

print('Volumen:', volumen, 'metros␣cúbicos.␣', end='')
print('Gracias␣por␣usar␣el␣programa.␣Adiós.')
```

La penúltima línea contiene un *print* con un argumento especial: «*end*=''». El mensaje de agradecimiento se mostrará ahora en la misma línea que el resultado del cálculo. Y una sutileza: hemos añadido un espacio en blanco tras el punto en la cadena `'metros␣cúbicos.'` para evitar que la siguiente línea se imprimiera pegada a ese punto. Este es el resultado de una ejecución del programa:

```
Programa para el cálculo del volumen de una esfera.
Dame el radio (en metros): 2↵
Volumen: 33.510321638291124 metros cúbicos. Gracias por usar el programa. Adiós.
```

Es el momento de recordar que podemos formatear una cadena para mostrar el resultado del cálculo:

esfera.py
```
from math import pi

print('Programa␣para␣el␣cálculo␣del␣volumen␣de␣una␣esfera.')

radio = float(input('Dame␣el␣radio␣(en␣metros):␣'))

volumen = 4 / 3 * pi * radio ** 3

print('Volumen␣{0:.2f}␣metros␣cúbicos.␣'.format(volumen), end='')
print('Gracias␣por␣usar␣el␣programa.␣Adiós.')
```

```
Programa para el cálculo del volumen de una esfera.
Dame el radio (en metros): 2↵
Volumen 33.51 metros cúbicos. Gracias por usar el programa. Adiós.
```

3.4. Sobre la legibilidad de los programas

Hemos visto cómo un uso apropiado de las líneas en blanco ayuda a hacer más legibles los programas. Vale la pena que abundemos en la cuestión de la legibilidad. Los programadores pasan muchas horas leyendo programas escritos por ellos mismos o por otros. Las razones son varias:

- Puede que hayas de seguir trabajando en un proyecto que abandonaste hace tiempo. Eso supone que releas lo que escribiste para continuar.

- O puede que un programa que ya habías dado por bueno se revele defectuoso unos días o meses después de ser utilizado a diario, con lo que deberás releer tu programa y buscar los errores cometidos para corregirlos.

- O también es posible que un programa ya escrito y sin defectos deba extenderse para que se le añada nueva funcionalidad o se adapte a nuevos estándares. Nuevamente te tocará releer buena parte del código para introducir los cambios necesarios.

- O quizá hayas conseguido trabajo en una empresa y te asignen la mejora de una pieza de código que escribió otro programador. Cuando hayas de estudiarlo, le estarás infinitamente agradecido si se preocupó de la legibilidad.

- O podría ser el caso de que estuvieses escribiendo programas para demostrar al profesorado que has alcanzado los objetivos de una asignatura de aprendizaje de la programación. Seguro que te gustará ver al evaluador contento cuando lea tus programas y sea capaz de entenderlos.

En todos estos casos (y en muchos otros), haberse asegurado de facilitar la lectura del código será un elemento con un claro impacto en la productividad.

3.4.1. Algunos convenios

Comparemos dos programas que hacen lo mismo desde el punto de vista de la legibilidad. Este es el primer programa:

ilegible.py
```
h = float(input('Dame h: '))
v = float(input('y v: '))
z = h * v
print('Resultado 1 {0:6.2f}'.format(z))
v = 2 * h + v + v
print('Resultado 2 {0:6.2f}'.format(v))
```

Y este es el segundo:

legible.py
```
print('Programa para el cálculo del perímetro y el área de un rectángulo.')

altura = float(input('Dame la altura (en metros): '))
anchura = float(input('Dame la anchura (en metros): '))

área = altura * anchura
perímetro = 2 * altura + 2 * anchura

print('El perímetro es de {0:6.2f} metros.'.format(perímetro))
print('El área es de {0:6.2f} metros cuadrados.'.format(área))
```

Basta con leer este segundo programa para saber qué hace. *Evidentemente*, el programa pide la altura y la anchura de un rectángulo y calcula su perímetro y área, valores que muestra a continuación. Son muchos los elementos que han ayudado a hacer más legible el segundo programa:

- `ilegible.py` usa nombres arbitrarios y breves para las variables, mientras que el programa `legible.py` *utiliza identificadores representativos y tan largos como sea necesario*. El programador de `ilegible.py` pensaba más en teclear poco que en hacer comprensible el programa. Además, `ilegible.py` usa una misma variable, *v*, para dos propósitos distintos: albergar primero una anchura y, después, un perímetro.

- `ilegible.py` no tiene una estructura clara: mezcla cálculos con impresión de resultados. En su lugar, `legible.py` *diferencia claramente zonas distintas del programa* (lectura

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

de datos, realización de cálculos y visualización de resultados) y llega a usar marcas visuales como las líneas en blanco para separarlas. Probablemente el programador de `ilegible.py` escribía el programa conforme se le iban ocurriendo cosas. El programador de `legible.py` tenía claro qué iba a hacer desde el principio: planificó la estructura del programa.

- `ilegible.py` utiliza fórmulas poco frecuentes para realizar algunos de los cálculos: la forma en que calcula el perímetro es válida, pero poco ortodoxa. Por contra, `legible.py` *utiliza formas de expresión de los cálculos que son estándar*. El programador de `ilegible.py` debería haber pensado en los convenios a la hora de utilizar fórmulas.

- Los mensajes de `ilegible.py`, tanto al pedir datos como al mostrar resultados, son de pésima calidad. Un usuario que se enfrenta al programa por primera vez tendrá serios problemas para entender qué se le pide y qué se le muestra como resultado. El programa `legible.py` *emplea mensajes de entrada/salida muy informativos*. Seguro que el programador de `ilegible.py` pensaba que él sería el único usuario de su programa.

Atenerte a las reglas usadas en `legible.py` será fundamental para hacer legibles tus programas.

▶ 42 Diseña un programa legible que solicite el radio de una circunferencia y muestre su área y perímetro con solo 2 decimales.

3.4.2. Comentarios

Dentro de poco empezaremos a realizar programas de mayor envergadura y con mucha mayor complicación. Incluso observando las reglas indicadas, va a resultar una tarea ardua leer un programa completo.

Un modo de aumentar la legibilidad de un programa consiste en intercalar *comentarios* que expliquen su finalidad o que aclaren sus pasajes más oscuros.

Como esos comentarios solo tienen por objeto facilitar la legibilidad de los programas para los programadores, pueden escribirse en el idioma que desees. Cuando el intérprete Python ve un comentario no hace nada con él: lo omite. ¿Cómo le indicamos al intérprete que cierto texto es un comentario? Necesitamos alguna marca especial. Los comentarios Python se inician con el símbolo # (que se lee «almohadilla»): todo texto desde la almohadilla hasta el final de la línea se considera comentario y, en consecuencia, es omitido por Python.

He aquí un programa con comentarios:

rectangulo.py

```
# Programa: rectangulo.py
# Propósito: Calcula el perímetro y el área de un rectángulo a partir de su altura y anchura.
# Autor: John Cleese
# Fecha: 1/1/2010

# Petición de los datos (en metros)
altura = float(input('Dame la altura (en metros): '))
anchura = float(input('Dame la anchura (en metros): '))

# Cálculo del área y del perímetro
área = altura * anchura
perímetro = 2 * altura + 2 * anchura

# Impresión de resultados por pantalla
print('El perímetro es de {0:6.2f} metros.'.format(perímetro)) # solo dos decimales.
print('El área es de {0:6.2f} metros cuadrados.'.format(área))
```

- en la cabecera del programa, comentando el nombre del programa, su propósito, el autor y la fecha;

- al principio de cada una de las «grandes zonas» del programa, indicando qué se hace en ellas;
- y al final de una de las líneas (la penúltima), para comentar alguna peculiaridad de la misma.

Es buena práctica que «comentes» tus programas. Pero ten presente que no hay reglas fijas que indiquen cuándo, dónde y cómo comentar los programas: las que acabes adoptando formarán parte de tu estilo de programación.

3.5. Gráficos de tortuga

Todos los programas que te hemos presentado utilizan el teclado y la pantalla en «modo texto» para interactuar con el usuario. Sin embargo, estás acostumbrado a interactuar con el ordenador mediante un terminal gráfico y usando, además del teclado, el ratón.

Python trae de serie una librería para la implementación de interfaces gráficas de usuario, esto es, ofrece a través de una librería la capacidad de crear ventanas, poblarlas con menús, botones, cajas de texto, etcétera, y definir el comportamiento de estos elementos al interactuar con el usuario. La librería se llama Tkinter. Es pronto para que nos enfrentemos a ella. En este libro nos limitaremos a crear aplicaciones gráficas muy sencillas con una «tortuga». ¿Una tortuga? El nombre de este tipo de gráficos tiene su historia y te damos algunas pinceladas en el cuadro «Logo y la tortuga».

Logo y la tortuga

Ha habido varios intentos de diseñar lenguajes que faciliten a los niños el aprendizaje de la programación. Seymour Papert y Wally Feurzeig diseñaron, a mediados de los años 60 del siglo XX, un lenguaje de programación muy sencillo que tenía ese objetivo. El lenguaje se denomina Logo y, ya entonces, tenía una fuerte orientación a los gráficos. Los sistemas informáticos de la época aún eran mastodónticos y su precio los hacía asequibles casi exclusivamente para grandes empresas y universidades. Pensar en enseñar a programar a los niños era toda una osadía.

Un elemento esencial de Logo era la posibilidad de confeccionar dibujos con un «plotter virtual». Se podía controlar un lápiz al que dar órdenes del estilo «avanza 100 pasos», «gira 45 grados a la derecha», «levanta el lápiz», «avanza 10 pasos», «baja el lápiz» y «avanza 100 pasos». El lápiz se representaba con un triángulo isósceles acutángulo con el vértice más agudo orientado en la dirección del movimiento. Para hacer más atractivo el sistema, se denominó «tortuga» al triángulo. De ahí que a los sistemas gráficos inspirados en la idea de Logo se les denomine genéricamente «sistemas gráficos de tortuga».

Imagina una tortuga que lleva un lápiz en la boca (¿?) y está esperando nuestras órdenes para dibujar sobre un gran papel extendido en el suelo. Le podemos dar órdenes sencillas, del tipo «apoya el lápiz en el papel», «avanza 100 pasos», «gira 10 grados a la derecha», «levanta el lápiz». Tan pronto recibe una orden, la tortuga la ejecuta. Si el lápiz está apoyado en la superficie, avanzar 100 pasos supone hacer una línea de esa longitud en la dirección hacia la que miraba la tortuga, dejando a la tortuga en una nueva posición. Si el lápiz no está apoyado, avanzar esos 100 pasos no tendrá un efecto visible sobre el papel, pero habrá desplazado igualmente a la tortuga en la dirección hacia la que miraba.

Escribamos un programa que haga avanzar 100 pasos a la tortuga y deje un trazo en pantalla. El programa tendrá cuatro partes:

- Primero se importarán los elementos necesarios del módulo *turtle*.
- Luego crearemos una pantalla a la que daremos un tamaño.
- Después crearemos una tortuga y le diremos que avance 100 pasos.
- Y finalmente detendremos el programa hasta que el usuario pulse el botón del ratón en la superficie de dibujo.

He aquí el programa:

```
grafico.py
from turtle import Screen, Turtle

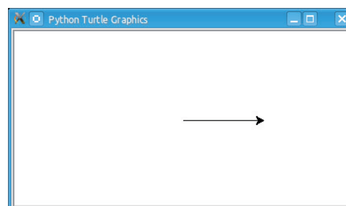
pantalla = Screen()
pantalla.setup(425, 225)
pantalla.screensize(400, 200)

tortuga = Turtle()
tortuga.forward(100)

pantalla.exitonclick()
```

Antes de ejecutarlo, veamos cómo hemos codificado cada una de las cuatro partes. Primero hemos importado los elementos *Screen* y *Turtle* definidos en la librería *turtle* con la sentencia **from** *turtle* **import** *Screen*, *Turtle*. A continuación, hemos creado una pantalla (en inglés, «screen») y la hemos almacenado en la variable *pantalla*. Las dos siguientes líneas han fijado la dimensión de la pantalla. La primera de ellas invoca al método *setup* de *pantalla*, que fija el ancho (425 píxeles) y alto (225 píxeles) de la ventana. La segunda invoca al método *screensize*, que fija el tamaño de la superficie de dibujo (400 píxeles de ancho y 200 de alto). Puedes observar que la ventana es algo más grande que la superficie de dibujo (unos 25 píxeles adicionales en cada dimensión): es porque la ventana contiene algunos elementos decorativos que necesitan su propio espacio. Después hemos creado una tortuga (en inglés, «turtle») y hemos almacenado una referencia a ella en la variable *tortuga*. La siguiente sentencia ejecuta sobre *tortuga* el método *forward* con el argumento 100. Le estamos dando una orden a la tortuga: que avance 100 pasos («forward», en inglés, significa «adelante») en la dirección en la que mira, que por defecto es hacia la derecha (o, si lo prefieres considerar en términos de puntos cardinales, hacia el este). La última sentencia contiene una llamada a un método de *pantalla* que evita que la ventana en la que se dibuja desaparezca inmediatamente: el método *exitonclick* fuerza a esperar a que el usuario haga clic en la ventana.

Ya podemos ejecutar el programa. En pantalla aparecerá esto:



El triángulo es la tortuga y la línea es el rastro que ha dejado al desplazarse. Pulsa en el interior de la ventana para que se cierre y finalice la ejecución del programa.

El método *left* hace que la tortuga gire hacia la izquierda tantos grados como se indique en el único argumento del método. Si combinamos *forward* y *left* podemos dibujar un cuadrado con unas pocas órdenes:

```
grafico.py
from turtle import Screen, Turtle

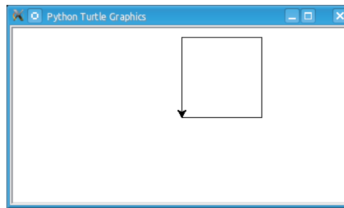
pantalla = Screen()
pantalla.setup(425, 225)
pantalla.screensize(400, 200)

tortuga = Turtle()
tortuga.forward(100)
tortuga.left(90)
tortuga.forward(100)
tortuga.left(90)
tortuga.forward(100)
```

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

```
tortuga.left(90)
tortuga.forward(100)

pantalla.exitonclick()
```



Además de *left*, que gira hacia la izquierda, tienes el método *right*, que gira hacia la derecha.

El lápiz puede levantarse para hacer que no deje rastro. Si no se pudiera, todos los dibujos estarían hechos con un solo trazo. El método *penup*, sin argumentos, levanta el lápiz, y el método *pendown*, también sin argumentos, lo vuelve a apoyar en la superficie de dibujo. Usamos ambos métodos para dibujar dos cuadrados no conectados:

grafico.py
```
from turtle import Screen, Turtle

pantalla = Screen()
pantalla.setup(425, 225)
pantalla.screensize(400, 200)

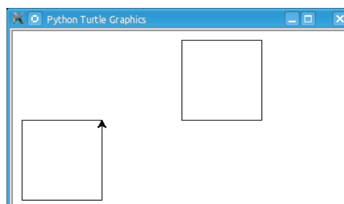
tortuga = Turtle()
tortuga.forward(100)
tortuga.left(90)
tortuga.forward(100)
tortuga.left(90)
tortuga.forward(100)
tortuga.left(90)
tortuga.forward(100)

tortuga.penup()
tortuga.right(90)
tortuga.forward(100)
tortuga.pendown()

tortuga.forward(100)
tortuga.left(90)
tortuga.forward(100)
tortuga.left(90)
tortuga.forward(100)
tortuga.left(90)
tortuga.forward(100)

pantalla.exitonclick()
```

Este es el resultado:



¿Sabes qué orden ha generado cada uno de los lados de ambos cuadrados? Fíjate en que el dibujo se hace a una velocidad ralentizada y es fácil ver cómo se va ejecutando cada orden.

Aquí relacionamos los métodos básicos de dibujo con la tortuga. Están todos los que ya hemos presentado y alguno más:

- *forward*(*d*): Avanza *d* pasos.
- *backward*(*d*): Retrocede *d* pasos.
- *left*(*g*): Gira a la izquierda *g* grados.
- *right*(*g*): Gira a la derecha *g* grados.
- *penup*(): Levanta el lápiz.
- *pendown*(): Baja el lápiz.

▶ 43 Diseña un programa que dibuje un triángulo equilátero con la tortuga.

▶ 44 Diseña un programa que dibuje un cuadrado cuyo lado mida 200 pasos y otro cuadrado de lado 100 centrado en su interior.

Podemos controlar algunos elementos del aspecto de las líneas, como el grosor del trazo y el color:

- *pensize*(*s*): Usa un lápiz con trazo de *s* píxeles de grosor.
- *pencolor*(*c*): Usa el color *c* para los trazos, donde *c* es una cadena con el nombre del color en inglés. Algunas cadenas de color válidas son `'white'`, `'black'`, `'red'`, `'blue'`, `'green'`, `'cyan'`, `'magenta'`, `'yellow'`, `'pink'` y `'orange'`.

Practica con los siguientes ejercicios.

▶ 45 Diseña un programa que dibuje un triángulo equilátero con la tortuga. El trazo del triángulo debe tener un grosor de 10 píxeles.

▶ 46 Diseña un programa que dibuje un cuadrado cuyo lado mida 200 pasos y otro cuadrado de lado 100 centrado en su interior. El cuadrado exterior ha de ser de color rojo y el interior de color azul.

Los métodos *forward*, *backward*, *left* y *right* permiten controlar a la tortuga con coordenadas y ángulos relativos a la posición y orientación de la tortuga. Si decimos dos veces *forward*(10), la tortuga habrá avanzado 20 pasos desde la posición de partida en la dirección a la que apunta. Si apunta al norte y damos dos órdenes consecutivas *left*(90), la tortuga apuntará al sur. En ocasiones querremos dar órdenes con coordenadas y ángulos absolutos. Estos otros métodos permiten controlar a la tortuga con valores absolutos:

- *goto*(*x*, *y*): Ubica a la tortuga en la posición de coordenadas (x, y).
- *setx*(*x*): Ubica a la tortuga en la posición de abcisa *x* y la misma ordenada actual.
- *sety*(*y*): Ubica a la tortuga en la posición de ordenada *y* y la misma abcisa actual.
- *setheading*(*g*): Hace que la tortuga apunte en dirección *g* grados (donde 0 grados es la dirección este).
- *towards*(*x*,*y*): Hace que la tortuga apunte en dirección al punto (x, y).
- *home*(): Ubica a la tortuga en la posición de coordenadas $(0, 0)$.

▶ 47 Diseña un programa que dibuje un triángulo equilátero con la tortuga. No uses los métodos *left* o *right*.

▶ 48 Diseña un programa que dibuje un cuadrado cuyo lado mida 200 pasos y otro cuadrado de lado 100 centrado en su interior. No uses los métodos *left* o *right*.

Hay un par de utilidades de la tortuga que nos darán un poco de juego: *dot* y *circle*.

- *dot*(*d*): Dibuja un punto de diámetro *d* centrado en la posición actual.
- *circle*(*r*): Dibuja un círculo de radio *r*. El círculo está centrado a *r* pasos a la izquierda de la tortuga.
- *write*(*t*): Escribe el texto de la cadena *t* en pantalla, en la posición actual de la tortuga.

Además, podemos establecer el sistema de coordenadas de la pantalla con el método *setworldcoordinates*:

- *setworldcoordinates*(*x1*, *y1*, *x2*, *y2*): Establece el sistema de coordenadas de la pantalla, donde $(x1, y1)$ representa el vértice inferior izquierdo y $(x2, y2)$ el vértice superior derecho.

Pongamos en práctica mucho de lo aprendido:

grafico.py
```python
from turtle import Screen, Turtle

pantalla = Screen()
pantalla.setup(425, 425)
pantalla.screensize(400, 400)
pantalla.setworldcoordinates(-50, -150, 350, 250)

tortuga = Turtle()

tortuga.pensize(3)
tortuga.dot(10)
tortuga.forward(100)
tortuga.dot(10)
tortuga.forward(100)
tortuga.dot(10)
tortuga.forward(100)
tortuga.dot(10)

tortuga.penup()
tortuga.goto(0, 100)
tortuga.pendown()

tortuga.pencolor('red')
tortuga.pensize(5)
tortuga.circle(20)
tortuga.forward(50)
tortuga.pensize(4)
tortuga.left(20)
tortuga.circle(20)
tortuga.forward(50)
tortuga.pensize(3)
tortuga.left(20)
tortuga.circle(20)
tortuga.forward(50)
tortuga.pensize(2)
tortuga.left(20)
tortuga.circle(20)
```

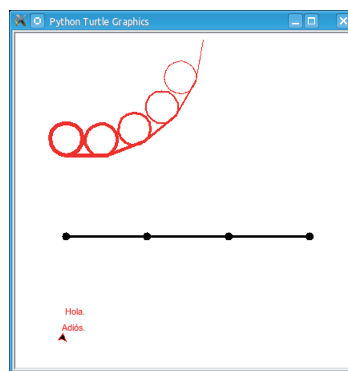
```
tortuga.forward(50)
tortuga.pensize(1)
tortuga.left(20)
tortuga.circle(20)
tortuga.forward(50)

tortuga.penup()
tortuga.goto(0, -100)
tortuga.towards(0, 0)

tortuga.write('Hola.')
tortuga.backward(20)
tortuga.write('Adiós.')

pantalla.exitonclick()
```

Este es el resultado:



Detengámonos a hacer una observación: el programa está repleto de fragmentos que se repiten una y otra vez. Fíjate en la zona en la que se dibuja la línea con puntos. Es una sucesión de fragmentos de la forma:

```
tortuga.dot(10)
tortuga.forward(100)
```

Y la zona en la que se dibujan los círculos es poco más que una sucesión de grupos de líneas como este:

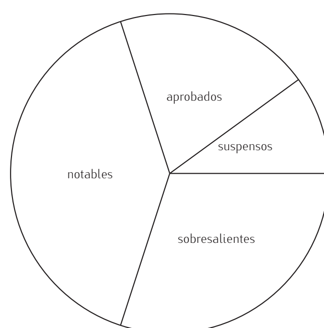
```
tortuga.pensize(x)
tortuga.left(20)
tortuga.circle(20)
tortuga.forward(50)
```

(donde *x* va tomando valores decrecientes de 5 a 1). Ha de haber modos de hacer que los programas sean más compactos. Tranquilo: los hay. Ya llegaremos.

Por otra parte, habrás comprobado que el dibujo aparece lentamente aunque tu ordenador sea potente y vemos a la tortuga ir desplazándose casi punto a punto por la pantalla. ¡Con razón la llaman «tortuga»! Ese efecto es deliberado y puedes evitarlo. El método *speed* permite controlar la velocidad con la que se desplaza la tortuga: su argumento es un número entre 0 y 10, donde 1 es la mínima velocidad y 10 es la máxima… siempre que desees que el movimiento de la tortuga no sea instantáneo. Si quieres que sea instantáneo, fija la velocidad al valor 0. Has de saber que la velocidad por defecto es 6. Las velocidades más lentas pueden venir bien para entender qué está ocurriendo cuando el programa no hace lo que uno cree que debiera hacer.

Vamos a hacer un primer programa con salida gráfica y que presente cierta utilidad: un programa que muestra el porcentaje de suspensos, aprobados, notables y sobresalientes de una asignatura mediante un «gráfico de pastel». He aquí un ejemplo de gráfico como el que deseamos

para una tasa de suspensos del 10%, un 20% de aprobados, un 40% de notables y un 30% de sobresalientes:



Diseñaremos nuestro programa paso a paso. Empezaremos por crear el círculo.

pastel.py
```
from turtle import Screen, Turtle

radio = 300

pantalla = Screen()
tortuga = Turtle()
tortuga.speed(0)

tortuga.penup()
tortuga.goto(0, -radio)
tortuga.pendown()
tortuga.circle(radio)
tortuga.penup()
tortuga.home()
tortuga.pendown()

pantalla.exitonclick()
```

Hemos empezado bajando el cursor *radio* unidades para dibujar a continuación un círculo. Como el círculo se dibuja a mano izquierda de la tortuga y esta mira hacia el este, el círculo se dibujará justo encima. La tortuga vuelve al punto original con *tortuga*.*home*(). Observa cómo hemos levantado y bajado el lápiz a conveniencia para asegurarnos de que solo se dibuja el círculo, y no cada uno de los movimientos de la tortuga. El radio se ha parametrizado almacenando su valor en una variable. Si más adelante deseamos cambiar el tamaño del círculo, será fácil: bastará con cambiar el valor de una sola variable.

Ahora vamos a asignar un porcentaje a cada una de las calificaciones. Por cierto: añadir unos comentarios mejorará la legibilidad del programa.

pastel.py
```
from turtle import Screen, Turtle

# Calificaciones
suspensos = 10
aprobados = 20
notables = 40
sobresalientes = 30

# Radio del círculo
radio = 300

# Inicialización
pantalla = Screen()
tortuga = Turtle()
```

```
tortuga.speed(0)

# Dibujo del círculo exterior.
tortuga.penup()
tortuga.goto(0, -radio)
tortuga.pendown()
tortuga.circle(radio)
tortuga.penup()
tortuga.home()
tortuga.pendown()

# Salir cuando se pulse el botón en la ventana.
pantalla.exitonclick()
```

Para dibujar cada línea divisoria entre porciones de la tarta tenemos que calcular el ángulo correspondiente. Y repetir el proceso para cada porción de la tarta:

pastel.py

```
from turtle import Screen, Turtle

# Calificaciones
suspensos = 10
aprobados = 20
notables = 40
sobresalientes = 30

# Radio del círculo
radio = 300

# Inicialización
pantalla = Screen()
tortuga = Turtle()
tortuga.speed(0)

# Dibujo del círculo exterior.
tortuga.penup()
tortuga.goto(0, -radio)
tortuga.pendown()
tortuga.circle(radio)
tortuga.penup()
tortuga.home()
tortuga.pendown()

# Dibujo de la línea para los suspensos.
ángulo =  360 * suspensos / 100
tortuga.left(ángulo)
tortuga.forward(radio)
tortuga.backward(radio)

# Escribir el texto para los suspensos.
tortuga.penup()
tortuga.right(ángulo / 2)
tortuga.forward(radio / 2)
tortuga.write('suspensos')
tortuga.backward(radio / 2)
tortuga.left(ángulo / 2)
tortuga.pendown()

# Dibujo de la línea para los aprobados.
ángulo =  360 * aprobados / 100
tortuga.left(ángulo)
```

```
tortuga.forward(radio)
tortuga.backward(radio)

# Escribir el texto para los aprobados.
tortuga.penup()
tortuga.right(ángulo / 2)
tortuga.forward(radio / 2)
tortuga.write('aprobados')
tortuga.backward(radio / 2)
tortuga.left(ángulo / 2)
tortuga.pendown()

# Dibujo de la línea para los notables.
ángulo =  360 * notables / 100
tortuga.left(ángulo)
tortuga.forward(radio)
tortuga.backward(radio)

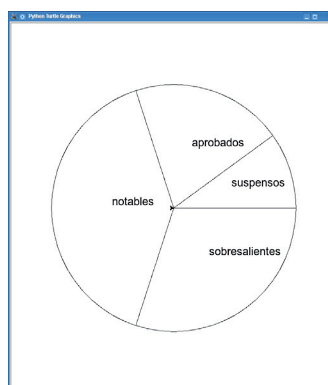
# Escribir el texto para los notables.
tortuga.penup()
tortuga.right(ángulo / 2)
tortuga.forward(radio / 2)
tortuga.write('notables')
tortuga.backward(radio / 2)
tortuga.left(ángulo / 2)
tortuga.pendown()

# Dibujo de la línea para los sobresalientes.
ángulo =  360 * sobresalientes / 100
tortuga.left(ángulo)
tortuga.forward(radio)
tortuga.backward(radio)

# Escribir el texto para los sobresalientes.
tortuga.penup()
tortuga.right(ángulo / 2)
tortuga.forward(radio / 2)
tortuga.write('sobresalientes')
tortuga.backward(radio / 2)
tortuga.left(ángulo / 2)
tortuga.pendown()

# Salir cuando se pulse el botón en la ventana.
pantalla.exitonclick()
```

Analiza bien cada uno de los bloques que componen el programa. Este es el resultado de la ejecución:



Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

Mmmm. Casi perfecto. Lo único que no queda bien es que la tortuga se queda en la pantalla y «ensucia» el resultado. Esto nos da pie para presentar dos métodos más:

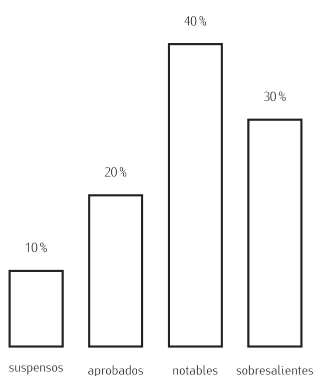
- *hideturtle*`()`: Esconde la tortuga.
- *showturtle*`()`: Muestra la tortuga.

Con esta información, modifica tú mismo el programa para que desaparezca la tortuga al final.

▶ 49 Modifica el programa para que sea el usuario quien proporcione, mediante el teclado, el valor del porcentaje de suspensos, aprobados, notables y sobresalientes.

▶ 50 Modifica el programa para que sea el usuario quien proporcione, mediante el teclado, el *número* de suspensos, aprobados, notables y sobresalientes. (Antes de dibujar el gráfico de pastel debes convertir esas cantidades en porcentajes).

▶ 51 Queremos representar la información de forma diferente: mediante un gráfico de barras. He aquí cómo:



Diseña un programa que solicite por teclado el número de personas con cada una de las cuatro calificaciones y muestre el resultado con un gráfico de barras.

Y antes de acabar, abundemos en la observación que hicimos antes: nuestro último programa consta de una serie de bloques básicamente idénticos. Cada una de las cuatro porciones repite una serie de órdenes en las que apenas cambia un valor numérico y una cadena. Ya veremos cómo reducir este programa usando funciones y estructuras de control. Empezaremos por las estructuras de control.

Capítulo 4

Estructuras de control

—De ahí que estén dando vueltas continuamente, supongo —dijo Alicia.
—Si, así es —dijo el Sombrerero—, conforme se van ensuciando las cosas.
—Pero ¿qué ocurre cuando vuelven al principio de nuevo? —se atrevió a preguntar Alicia.

Alicia en el país de las maravillas, Lewis Carroll

Los programas que hemos aprendido a construir hasta el momento presentan siempre una misma secuencia de acciones:

1) Se piden datos al usuario (asignando a variables valores obtenidos con *input*).

2) Se efectúan cálculos con los datos introducidos por el usuario, guardando el resultado en variables (mediante asignaciones).

3) Se muestran por pantalla los resultados almacenados en variables (mediante la función *print*).

Estos programas se forman como una serie de líneas que se ejecutan una tras otra, desde la primera hasta la última y siguiendo el mismo orden con el que aparecen en el fichero: el *flujo de ejecución* del programa es estrictamente secuencial.

No obstante, es posible alterar el flujo de ejecución de los programas para hacer que:

- tomen decisiones a partir de los datos y/o resultados intermedios y, en función de estas, ejecuten ciertas sentencias y otras no;
- tomen decisiones a partir de los datos y/o resultados intermedios y, en función de estas, ejecuten ciertas sentencias más de una vez.

El primer tipo de alteración del flujo de control se efectúa con *sentencias condicionales* o *de selección* y el segundo tipo con *sentencias iterativas* o *de repetición*. Las sentencias que permiten alterar el flujo de ejecución se engloban en las denominadas *estructuras de control de flujo* (que abreviamos con el término «estructuras de control»).

Estudiaremos una forma adicional de alterar el flujo de control que permite señalar, detectar y tratar los errores que se producen al ejecutar un programa: las sentencias de emisión y captura de excepciones.

4.1. Sentencias condicionales

4.1.1. Un programa ilustrativo: resolución de ecuaciones de primer grado

Veamos un ejemplo. Diseñemos un programa para resolver cualquier ecuación de primer grado de la forma

$$ax + b = 0,$$

donde x es la incógnita.

Antes de empezar hemos de responder a dos preguntas:

1) ¿Cuáles son los datos del problema? (Generalmente, los datos del problema se pedirán al usuario con *input*).

   En nuestro problema, los coeficientes a y b son los datos del problema.

2) ¿Qué deseamos calcular? (Típicamente, acabaremos mostrando al usuario su valor mediante una llamada a *print*).

   Obviamente, el valor de x.

Ahora que conocemos los *datos de entrada* y el resultado que hemos de calcular, es decir, los *datos de salida*, nos preguntamos: ¿cómo calculamos la salida a partir de la entrada? En nuestro ejemplo, despejando x de la ecuación llegamos a la conclusión de que x se obtiene calculando $-b/a$.

Siguiendo el esquema de los programas que sabemos hacer, procederemos así:

1) Pediremos el valor de a y el valor de b (que supondremos de tipo flotante).

2) Calcularemos el valor de x como $-b/a$.

3) Mostraremos por pantalla el valor de x.

Empecemos creando un nuevo proyecto Pydev al que denominaremos **ecuaciones**. En su carpeta **src** crearemos un nuevo módulo llamado **primer_grado**:

```
primer_grado.py
print('Programa para la resolución de la ecuación a x + b = 0.')

a = float(input('Valor de a: '))
b = float(input('Valor de b: '))

x = -b / a

print('Solución: ', x)
```

Las líneas se ejecutan en el mismo orden con el que aparecen en el programa. Veámoslo funcionar:

```
Programa para la resolución de la ecuación a x + b = 0.
Valor de a: 10↲
Valor de b: 2↲
Solución:  -0.2
```

▶ 52 Un programador propone el siguiente programa para resolver la ecuación de primer grado:

```
primer_grado.py
print('Programa para la resolución de la ecuación a x + b = 0.')

a = float(input('Valor de a: '))
b = float(input('Valor de b: '))

a * x + b = 0

print('Solución: ', x)
```

¿Es correcto este programa? Si no, explica qué está mal.

▶ 53 Otro programador propone este programa:

```
primer_grado.py
print('Programa para la resolución de la ecuación a x + b = 0.')

```

```
x = -b / a

a = float(input('Valor de a: '))
b = float(input('Valor de b: '))

print('Solución: ', x)
```

¿Es correcto? Si no lo es, explica qué está mal.

Nuestro programa presenta un punto débil: cuando *a* vale 0, se produce un error de división por cero:

```
Programa para la resolución de la ecuación a x + b = 0.
Valor de a: 0
Valor de b: 3
Traceback (most recent call last):
  File "primer_grado.py", line 6, in <module>
    x = -b / a
ZeroDivisionError: float division by zero
```

Hemos de evitar los errores en tiempo de ejecución: detienen abruptamente la ejecución del programa y muestran mensajes de error poco comprensibles para el usuario del programa. Si al escribir el programa hemos previsto una solución para todo posible error de ejecución, podemos (y debemos) tomar el control de la situación en todo momento. Y si no lo hemos hecho, somos unos malos programadores[1].

Errores de ejecución

Hemos dicho que conviene evitar los errores de programa que se producen en tiempo de ejecución y, ciertamente, la industria de desarrollo de software realiza un gran esfuerzo para que sus productos estén libres de errores de ejecución. No obstante, el gran tamaño de los programas y su complejidad (unidos a las prisas por sacar los productos al mercado) hacen que muchos de estos errores acaben haciendo acto de presencia. Todos hemos sufrido la experiencia de, ejecutando una aplicación, obtener un mensaje de error indicando que se ha abortado la ejecución del programa o, peor aún, el computador se ha quedado «colgado». Si la aplicación contenía datos de trabajo importantes y no los habíamos guardado en disco, estos se habrán perdido irremisiblemente. Nada hay más irritante para el usuario que una aplicación poco estable, es decir, propensa a la comisión de errores en tiempo de ejecución.

El sistema operativo es, también, software, y está sujeto a los mismos problemas de desarrollo de software que las aplicaciones convencionales. Sin embargo, los errores en el sistema operativo son, por regla general, más graves, pues suelen ser estos los que dejan «colgado» al ordenador.

El famoso «sal y vuelve a entrar en la aplicación» o «reinicia el computador» que suele proponerse como solución práctica a muchos problemas de estos es consecuencia de los bajos niveles de calidad de buena parte del software que se comercializa.

4.1.2. La sentencia condicional if

En nuestro programa de ejemplo nos gustaría *detectar* si *a* vale cero para, *en ese caso*, no ejecutar el cálculo de la sexta línea de `primer_grado.py`, que es la que provoca el error. ¿Cómo hacer que cierta parte del programa se ejecute o deje de hacerlo en función de una condición determinada?

Los lenguajes de programación convencionales presentan una sentencia especial cuyo significado es:

«Al llegar a este punto, ejecuta esta(s) acción(es) *solo si* esta condición *es cierta*.»

[1]Aunque lo cierto es que programar es una tarea muy ardua y resulta muy difícil pensar en todo aquello que podría ir mal y anticiparse. Muchos errores de programación no se descubren hasta bien tarde. Seguro que, desgraciadamente, te ha tocado sufrirlo como usuario en más de un programa.

Este tipo de sentencia se denomina *condicional* o *de selección* y en Python es de la siguiente forma:

```
if condición:
    acción
    acción
    ...
    acción
```

(En inglés «if» significa «*si*»). Las acciones, que serán sentencias Python válidas, se escriben con un sangrado mayor que el de la línea que contiene la condición. Estas acciones solo se ejecutan si la condición proporciona como resultado el valor booleano True.

En nuestro caso, deseamos detectar la condición «*a* no vale 0» y, solo en ese caso, ejecutar las últimas líneas del programa:

```
primer_grado.py
1 print('Programa␣para␣la␣resolución␣de␣la␣ecuación␣a␣x␣+␣b␣=␣0.')
2
3 a = float(input('Valor␣de␣a:␣'))
4 b = float(input('Valor␣de␣b:␣'))
5
6 if a != 0:
7     x = -b / a
8     print('Solución:␣', x)
```

Analicemos detenidamente las líneas 6, 7 y 8. En la línea 6 aparece la palabra reservada if seguida de lo que, según hemos dicho, debe ser una condición. La condición se lee fácilmente si sabemos que != significa «es distinto de». Así pues, la línea 6 se lee «si *a* es distinto de 0». La línea que empieza con if debe finalizar obligatoriamente con dos puntos (:). Fíjate en que las dos siguientes líneas se escriben más a la derecha[2]. Decimos que estas líneas presentan mayor *sangrado* o *indentación* que la línea que empieza con if. Este mayor sangrado indica que la ejecución de estas dos líneas depende de que se satisfaga la condición *a* != 0: solo cuando esta es cierta se ejecutan las líneas de mayor sangrado. Así pues, cuando *a* valga 0, esas líneas *no se ejecutarán*, evitando de este modo el error de división por cero.

Veamos qué ocurre ahora si volvemos a introducir los datos que antes provocaron el error:

```
Programa para la resolución de la ecuación a x + b = 0.
Valor de a: 0↵
Valor de b: 3↵
```

Mmmm... no ocurre nada. No se produce un error, es cierto, pero el programa acaba sin proporcionar ninguna información. Analicemos la causa. Las cinco primeras líneas del programa se han ejecutado (imprime un mensaje y nos pide los valores de *a* y *b*); también se ha ejecutado la sexta línea, pero dado que la condición no se ha cumplido (*a* vale 0), las líneas 7 y 8 se han ignorado y, como no hay más líneas en el programa, la ejecución ha finalizado sin más. No se ha producido un error, ciertamente, pero acabar así la ejecución del programa puede resultar un tanto confuso para el usuario.

Veamos qué hace este otro programa:

```
primer_grado.py
1 print('Programa␣para␣la␣resolución␣de␣la␣ecuación␣a␣x␣+␣b␣=␣0.')
2
3 a = float(input('Valor␣de␣a:␣'))
4 b = float(input('Valor␣de␣b:␣'))
5
6 if a != 0:
7     x = -b / a
8     print('Solución:␣', x)
9
```

[2]Para destacar esta característica, hemos dibujado una línea vertical que marca el nivel al que apareció el if.

```
10 if a == 0:
11     print('La␣ecuación␣no␣tiene␣solución.')
```

La línea 10 contiene, nuevamente, una sentencia condicional que afecta a la línea 11 (observa que está más sangrada). En lugar de `!=`, el operador de comparación utilizado ahora es `==`. La sentencia se lee «si a es igual a 0».

La ejecución con los mismos datos de antes es, ahora, un poco más clara:

```
Programa para la resolución de la ecuación a x + b = 0.
Valor de a: 0↵
Valor de b: 3↵
La ecuación no tiene solución.
```

Ante datos tales que a es distinto de 0, el programa resuelve la ecuación:

```
Programa para la resolución de la ecuación a x + b = 0.
Valor de a: 1↵
Valor de b: -1↵
Solución:  1.0
```

Estudiemos con detenimiento qué ha pasado en cada uno de los casos:

$a = 0$ y $b = 3$	$a = 1$ y $b = -1$
Las líneas 1, 3 y 4 se ejecutan, con lo que se imprime un mensaje y se leen los valores de a y b.	Las líneas 1, 3 y 4 se ejecutan, con lo que se imprime un mensaje y se leen los valores de a y b.
La línea 6 se ejecuta y el resultado de la comparación es *falso*.	La línea 6 se ejecuta y el resultado de la comparación es *cierto*.
Las líneas 7 y 8 se ignoran.	Se ejecutan las líneas 7 y 8, con lo que se muestra por pantalla el valor de la Solución: **`Solución: 1`**.
La línea 10 se ejecuta y el resultado de la comparación es *cierto*.	La línea 10 se ejecuta y el resultado de la comparación es *falso*.
La línea 11 se ejecuta y se muestra por pantalla el mensaje **`La ecuación no tiene solución.`**	La línea 11 se ignora.

Este tipo de análisis, en el que seguimos el curso del programa línea a línea para una configuración dada de los datos de entrada, recibe el nombre de *traza* de ejecución. Las *trazas* de ejecución son de gran ayuda para comprender qué hace un programa y localizar así posibles errores.

▶ 54 Un estudiante ha tecleado el último programa así:

primer_grado.py
```
1  print('Programa␣para␣la␣resolución␣de␣la␣ecuación␣a␣x␣+␣b␣=␣0.')
2
3  a = float(input('Valor␣de␣a:␣'))
4  b = float(input('Valor␣de␣b:␣'))
5
6  if a != 0:
7      x = -b / a
8      print('Solución:␣', x)
9
10 if a = 0:
11     print('La␣ecuación␣no␣tiene␣solución.')
```

Al ejecutarlo obtiene este error:

```
  File "primer_grado.py", line 10
    if a = 0:
         ^
SyntaxError: invalid syntax
```

Por más que el estudiante lee el programa, no encuentra fallo alguno. Él dice que la línea 10, la marcada como errónea, se lee así: «si *a* es igual a cero…». ¿Está en lo cierto? ¿Por qué se detecta un error?

▶ 55 Un programador primerizo cree que la línea 10 de la última versión del programa **primer_grado.py** es innecesaria, así que propone esta otra versión como solución válida:

primer_grado.py
```
print('Programa para la resolución de la ecuación a x + b = 0.')

a = float(input('Valor de a: '))
b = float(input('Valor de b: '))

if a != 0:
    x = -b / a
    print('Solución: ', x)

print('La ecuación no tiene solución.')
```

Haz una traza del programa para $a = 2$ y $b = 2$. ¿Son correctos todos los mensajes que muestra por pantalla el programa?

4.1.3. Sentencias condicionales anidadas

Vamos a realizar un último refinamiento del programa. De momento, cuando *a* es 0 el programa muestra un mensaje que indica que la ecuación no tiene solución. Nosotros sabemos que esto no es cierto: si, además, *b* vale 0, entonces la ecuación tiene infinitas soluciones. Para que el programa dé una información correcta vamos a modificarlo de modo que, cuando *a* sea 0, muestre un mensaje u otro en función del valor de *b*:

primer_grado.py
```
1  print('Programa para la resolución de la ecuación a x + b = 0.')
2
3  a = float(input('Valor de a: '))
4  b = float(input('Valor de b: '))
5
6  if a != 0:
7      x = -b / a
8      print('Solución: ', x)
9
10 if a == 0:
11     if b != 0:
12         print('La ecuación no tiene solución.')
13     if b == 0:
14         print('La ecuación tiene infinitas soluciones.')
```

Fíjate en el sangrado de las líneas. Las líneas 11–14 están más a la derecha que la línea 10. Ninguna de ellas se ejecutará a menos que la condición de la línea 10 se satisfaga. Más aún, la línea 11 está más a la derecha que la línea 10, por lo que su ejecución depende del resultado de la condición de dicha línea; y la ejecución de la línea 12 depende de la satisfacción de la condición de la línea 11. Recuerda que *en los programas Python el sangrado determina de qué sentencia depende cada bloque* de sentencias.

Pues bien, acabamos de presentar una nueva idea muy potente: las estructuras de control pueden *anidarse*, es decir, aparecer unas «dentro» de otras. Esto no ha hecho más que empezar.

▶ 56 Indica qué líneas del último programa (y en qué orden) se ejecutarán para cada uno de los siguientes casos:

1) $a = 2$ y $b = 6$.

2) $a = 0$ y $b = 3$.

3) $a = 0$ y $b = -3$.

4) $a = 0$ y $b = 0$.

▶ 57 Diseña un programa que lea un número flotante por teclado y muestre por pantalla el mensaje «`El número es negativo.`» solo si el número es menor que cero.

▶ 58 Diseña un programa que lea un número flotante por teclado y muestre por pantalla el mensaje «`El número es positivo.`» solo si el número es mayor o *igual* que cero.

▶ 59 Diseña un programa que lea la edad de dos personas y diga quién es más joven, la primera o la segunda. Ten en cuenta que ambas pueden tener la misma edad. En tal caso, hazlo saber con un mensaje adecuado.

▶ 60 Diseña un programa que lea un carácter de teclado y muestre por pantalla el mensaje «Es paréntesis» solo si el carácter leído es un paréntesis abierto o cerrado.

▶ 61 Indica en cada uno de los siguientes programas qué valores en las respectivas entradas provocan la aparición de los distintos mensajes. Piensa primero la solución y comprueba luego que es correcta ayudándote con el ordenador.

1) misterio.py

```python
letra = input('Dame una letra minúscula: ')

if letra <= 'k':
    print('Es de las primeras del alfabeto')
if letra >= 'l':
    print('Es de las últimas del alfabeto')
```

2) cuadrante.py

```python
from math import floor # La función floor redondea hacia abajo.

grados = float(input('Dame un ángulo (en grados): '))

cuadrante = floor(grados) % 360 // 90
if cuadrante == 0:
    print('primer cuadrante')
if cuadrante == 1:
    print('segundo cuadrante')
if cuadrante == 2:
    print('tercer cuadrante')
if cuadrante == 3:
    print('cuarto cuadrante')
```

▶ 62 ¿Qué mostrará por pantalla el siguiente programa?

comparaciones.py

```python
if 14 < 120:
    print('Primer saludo')
```

```
if '14' < '120':
    print('Segundo saludo')
```

Por lo visto hasta el momento, podemos comparar valores numéricos con valores numéricos y cadenas con cadenas. Tanto los valores numéricos como las cadenas pueden ser el resultado de una expresión que aparezca explícitamente en la propia comparación. Por ejemplo, para saber si el producto de dos números enteros es igual a 100, podemos utilizar este programa:

es_cien.py
```
n = int(input('Dame un número: '))
m = int(input('Dame otro número: '))

if n * m == 100:
    print('El producto {0} * {1} es igual a 100'.format(n, m))
if n * m != 100:
    print('El producto {0} * {1} es distinto de 100'.format(n, m))
```

▶ 63 Diseña un programa que, dado un número entero, muestre por pantalla el mensaje «`El número es par.`» cuando el número sea par y el mensaje «`El número es impar.`» cuando sea impar.

(Una pista: un número es par si el resto de dividirlo por 2 es 0, e impar en caso contrario).

▶ 64 Diseña un programa que, dado un número entero, determine si este es el doble de un número impar. (Ejemplo: 14 es el doble de 7, que es impar).

▶ 65 Diseña un programa que, dados dos números enteros, muestre por pantalla uno de estos mensajes: «`El segundo es el cuadrado del primero.`», «`El segundo es menor que el cuadrado del primero.`» o bien «`El segundo es mayor que el cuadrado del primero.`», dependiendo de la verificación de la condición correspondiente al significado de cada mensaje.

▶ 66 Un capital de C euros a un interés del x por cien anual durante n años se convierte en $C \cdot (1 + x/100)^n$ euros. Diseña un programa Python que solicite la cantidad C, el interés x y el número de años n y calcule el capital final *solo si x es una cantidad positiva*.

▶ 67 Realiza un programa que calcule el desglose mínimo en billetes y monedas de una cantidad exacta de euros. Hay billetes de 500, 200, 100, 50, 20, 10 y 5 € y monedas de 2 y 1 €.

Por ejemplo, si deseamos conocer el desglose de 434 €, el programa mostrará por pantalla el siguiente resultado:

```
2 billetes de 200 euros.
1 billete de 20 euros.
1 billete de 10 euros.
2 monedas de 2 euros.
```

(¿Que cómo se efectúa el desglose mínimo? Muy fácil. Empieza por calcular la división entera entre la cantidad y 500 (el valor de la mayor moneda): 434 entre 500 da 0, así que no hay billetes de 500 € en el desglose; divide a continuación la cantidad 434 entre 200, cabe a 2 y sobran 34, así que en el desglose hay 2 billetes de 200 €; dividimos a continuación 34 entre 100 y vemos que no hay ningún billete de 100 € en el desglose (cabe a 0); como el resto de la última división es 34, pasamos a dividir 34 entre 20 y vemos que el desglose incluye un billete de 20 € y aún nos faltan 14 € por desglosar...).

4.1.4. Otro ejemplo: resolución de ecuaciones de segundo grado

Para afianzar los conceptos presentados (y aprender alguno nuevo), vamos a presentar otro ejemplo. En esta ocasión vamos a resolver ecuaciones de segundo grado, que son de la forma

$$ax^2 + bx + c = 0.$$

¿Cuáles son los datos del problema? Los coeficientes a, b y c. ¿Qué deseamos calcular? Los valores de x que hacen cierta la ecuación. Dichos valores son:

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \qquad \text{y} \qquad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}.$$

Un programa directo para este cálculo es:

segundo_grado.py
```
from math import sqrt   # La función sqrt calcula la raíz cuadrada de un número.

print('Programa␣para␣la␣resolución␣de␣la␣ecuación␣a␣x*x␣+␣b␣x␣+␣c=␣0.')

a = float(input('Valor␣de␣a:␣'))
b = float(input('Valor␣de␣b:␣'))
c = float(input('Valor␣de␣c:␣'))

x1 = (-b + sqrt(b**2 - 4*a*c)) / (2 * a)
x2 = (-b - sqrt(b**2 - 4*a*c)) / (2 * a)

print('Soluciones:␣x1={0:.3f}␣y␣x2={1:.3f}'.format(x1, x2))
```

Ejecutemos el programa:

```
Programa para la resolución de la ecuación a x*x + b x + c= 0.
Valor de a: 2↵
Valor de b: 7↵
Valor de c: 2↵
Soluciones: x1=-0.314 y x2=-3.186
```

Un problema evidente de nuestro programa es la división por cero que tiene lugar cuando a vale 0 (pues entonces el denominador, $2a$, es nulo). Tratemos de evitar el problema de la división por cero del mismo modo que antes, pero mostrando un mensaje distinto, pues cuando a vale 0 la ecuación no es de segundo grado, sino de primer grado.

segundo_grado.py
```
from math import sqrt   # La función sqrt calcula la raíz cuadrada de un número.

print('Programa␣para␣la␣resolución␣de␣la␣ecuación␣a␣x*x␣+␣b␣x␣+␣c=␣0.')

a = float(input('Valor␣de␣a:␣'))
b = float(input('Valor␣de␣b:␣'))
c = float(input('Valor␣de␣c:␣'))

if a != 0:
    x1 = (-b + sqrt(b**2 - 4*a*c)) / (2 * a)
    x2 = (-b - sqrt(b**2 - 4*a*c)) / (2 * a)
    print('Soluciones:␣x1={0:.3f}␣y␣x2={1:.3f}'.format(x1, x2))

if a == 0:
    print('No␣es␣una␣ecuación␣de␣segundo␣grado.')
```

4.1.5. En caso contrario (else)

Fíjate en que tanto en el ejemplo que estamos desarrollando ahora como en el anterior, hemos recurrido a sentencias condicionales que conducen a ejecutar una acción si se cumple una condición y a ejecutar otra si esa misma condición no se cumple:

```
if condición:
    acciones
if condición contraria:
    otras acciones
```

Este tipo de combinación es muy frecuente, hasta el punto de que se ha incorporado al lenguaje de programación una forma abreviada que significa lo mismo:

```
if condición:
    acciones
else:
    otras acciones
```

La palabra «else» significa, en inglés, «si no» o «en caso contrario». Es muy importante que respetes el sangrado: las acciones siempre un poco a la derecha, y el **if** y el **else**, alineados en la misma columna.

segundo_grado.py
```python
from math import sqrt  # La función sqrt calcula la raíz cuadrada de un número.

print('Programa para la resolución de la ecuación a x*x + b x + c= 0.')

a = float(input('Valor de a: '))
b = float(input('Valor de b: '))
c = float(input('Valor de c: '))

if a != 0:
    x1 = (-b + sqrt(b**2 - 4*a*c)) / (2 * a)
    x2 = (-b - sqrt(b**2 - 4*a*c)) / (2 * a)
    print('Soluciones: x1={0:.3f} y x2={1:.3f}'.format(x1, x2))
else:
    print('No es una ecuación de segundo grado.')
```

El programa no acaba de estar bien. Es verdad que cuando *a* vale 0, la ecuación es de primer grado, pero, aunque sabemos resolverla, no lo estamos haciendo. Sería mucho mejor si, en ese caso, el programa nos ofreciera la solución:

segundo_grado.py
```python
from math import sqrt  # La función sqrt calcula la raíz cuadrada de un número.

print('Programa para la resolución de la ecuación a x*x + b x + c= 0.')

a = float(input('Valor de a: '))
b = float(input('Valor de b: '))
c = float(input('Valor de c: '))

if a != 0:
    x1 = (-b + sqrt(b**2 - 4*a*c)) / (2 * a)
    x2 = (-b - sqrt(b**2 - 4*a*c)) / (2 * a)
    print('Soluciones: x1={0:.3f} y x2={1:.3f}'.format(x1, x2))
else:
    x = -c / b
    print('Solución: x={0:.3f}'.format(x))
```

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

Mmmm... aún hay un problema: ¿Qué pasa si *a* vale 0 y *b* también vale 0? La secuencia de líneas que se ejecutará será: 1, 3, 5, 6, 7, 9 y 14. De la línea 14 no pasará porque se producirá una división por cero.

```
Programa para la resolución de la ecuación a x*x + b x + c= 0.
Valor de a: 0↵
Valor de b: 0↵
Valor de c: 2↵
Traceback (most recent call last):
  File "segundo_grado.py", line 14, in <module>
    x = -c / b
ZeroDivisionError: float division by zero
```

¿Cómo evitar este nuevo error? Muy sencillo, añadiendo nuevos controles con la sentencia **if**, tal y como hicimos para resolver correctamente una ecuación de primer grado:

segundo_grado.py

```
1  from math import sqrt   # La función sqrt calcula la raíz cuadrada de un número.
2
3  print('Programa␣para␣la␣resolución␣de␣la␣ecuación␣a␣x*x␣+␣b␣x␣+␣c=␣0.')
4
5  a = float(input('Valor␣de␣a:␣'))
6  b = float(input('Valor␣de␣b:␣'))
7  c = float(input('Valor␣de␣c:␣'))
8
9  if a != 0:
10     x1 = (-b + sqrt(b**2 - 4*a*c)) / (2 * a)
11     x2 = (-b - sqrt(b**2 - 4*a*c)) / (2 * a)
12     print('Soluciones:␣x1={0:.3f}␣y␣x2={1:.3f}'.format(x1, x2))
13 else:
14     if b != 0:
15         x = -c / b
16         print('Solución:␣x={0:.3f}'.format(x))
17     else:
18         if c != 0:
19             print('La␣ecuación␣no␣tiene␣solución.')
20         else:
21             print('La␣ecuación␣tiene␣infinitas␣soluciones.')
```

Es muy importante que te fijes en que las líneas 14–21 presentan un sangrado tal que *todas ellas* dependen del **else** de la línea 13. Las líneas 15 y 16 dependen del **if** de la línea 14, y las líneas 18–21 dependen del **else** de la línea 17. Estudia bien el programa: aparecen sentencias condicionales anidadas en otras sentencias condicionales que, a su vez, están anidadas. ¿Complicado? No tanto. Los principios que aplicamos son siempre los mismos. Si analizas el programa y lo estudias por partes, verás que no es *tan* difícil de entender. Pero quizá lo verdaderamente difícil no sea entender programas con bastantes niveles de anidamiento, sino diseñarlos.

▶ 68 ¿Hay alguna diferencia entre el programa anterior y este otro cuando los ejecutamos?

segundo_grado.py

```
from math import sqrt   # La función sqrt calcula la raíz cuadrada de un número.

print('Programa␣para␣la␣resolución␣de␣la␣ecuación␣a␣x*x␣+␣b␣x␣+␣c=␣0.')

a = float(input('Valor␣de␣a:␣'))
b = float(input('Valor␣de␣b:␣'))
c = float(input('Valor␣de␣c:␣'))

if a == 0:
    if b == 0:
        if c == 0:
            print('La␣ecuación␣tiene␣infinitas␣soluciones.')
        else:
```

```
            print('La␣ecuación␣no␣tiene␣solución.')
    else:
        x = -c / b
        print('Solución:␣x={0:.3f}'.format(x))
else:
    x1 = (-b + sqrt(b**2 - 4*a*c)) / (2 * a)
    x2 = (-b - sqrt(b**2 - 4*a*c)) / (2 * a)
    print('Soluciones:␣x1={0:.3f}␣y␣x2={1:.3f}'.format(x1, x2))
```

▶ 69 ¿Hay alguna diferencia entre el programa anterior y este otro cuando los ejecutamos?

segundo_grado.py

```
from math import sqrt  # La función sqrt calcula la raíz cuadrada de un número.

print('Programa␣para␣la␣resolución␣de␣la␣ecuación␣a␣x*x␣+␣b␣x␣+␣c=␣0.')

a = float(input('Valor␣de␣a:␣'))
b = float(input('Valor␣de␣b:␣'))
c = float(input('Valor␣de␣c:␣'))

if a == 0 and b == 0 and c == 0:
    print('La␣ecuación␣tiene␣infinitas␣soluciones.')
else:
    if a == 0 and b == 0:
        print('La␣ecuación␣no␣tiene␣solución.')
    else:
        if a == 0:
            x = -c / b
            print('Solución:␣x={0:.3f}'.format(x))
        else:
            x1 = (-b + sqrt(b**2 - 4*a*c)) / (2 * a)
            x2 = (-b - sqrt(b**2 - 4*a*c)) / (2 * a)
            print('Soluciones:␣x1={0:.3f}␣y␣x2={1:.3f}'.format(x1, x2))
```

▶ 70 Diseña un programa Python que lea un carácter cualquiera desde el teclado, y muestre el mensaje «`Es una MAYÚSCULA`» cuando el carácter sea una letra mayúscula y el mensaje «`Es una MINÚSCULA`» cuando sea una minúscula. En cualquier otro caso, no mostrará mensaje alguno. (Considera únicamente letras del alfabeto inglés). Pista: aunque parezca una obviedad, recuerda que una letra es minúscula si está entre la `'a'` y la `'z'`, y mayúscula si está entre la `'A'` y la `'Z'`.

▶ 71 Amplía la solución al ejercicio anterior para que cuando el carácter introducido no sea una letra muestre el mensaje «`No es una letra`». (Nota: no te preocupes por las letras eñe, ce cedilla, vocales acentuadas, etc.).

▶ 72 Amplía el programa del ejercicio anterior para que pueda identificar las letras eñe minúscula y mayúscula.

▶ 73 Modifica el programa que propusiste como solución al ejercicio 65 sustituyendo todas las condiciones que sea posible por cláusulas **else** de condiciones anteriores.

4.1.6. Una estrategia de diseño: refinamientos sucesivos

Es lógico que cuando estés aprendiendo a programar te cueste gran esfuerzo construir mentalmente un programa tan complicado, pero posiblemente sea porque sigues una aproximación equivocada: no debes intentar construir mentalmente *todo* el programa de una vez. Es recomendable que sigas una estrategia similar a la que hemos usado al desarrollar los programas de ejemplo:

Índice

1) Primero haz una versión sobre papel que resuelva el problema de forma directa y, posiblemente, un tanto tosca. Una buena estrategia es plantearse uno mismo el problema con unos datos concretos, resolverlo a mano con lápiz y papel y hacer un esquema con el orden de las operaciones realizadas y las decisiones tomadas. Tu primer programa puede pedir los datos de entrada (con *input*), hacer los cálculos del mismo modo que tú los hiciste sobre el papel (utilizando variables para los resultados intermedios, si fuera menester) y mostrar finalmente el resultado del cálculo (con *print*).

2) Analiza tu programa y considera si realmente resuelve el problema planteado: ¿es posible que se cometan errores en tiempo de ejecución?, ¿hay configuraciones de los datos que son especiales y, para ellas, el cálculo debe ser diferente?

3) Cada vez que te plantees una de estas preguntas y tengas una respuesta, modifica el programa en consecuencia. No hagas más de un cambio cada vez.

4) Si el programa ya funciona correctamente *para todas las entradas posibles* y eres capaz de anticiparte a los posibles errores de ejecución, ¡enhorabuena!, ya casi has terminado. En caso contrario, vuelve al paso 2.

5) Ahora que ya estás «seguro» de que todo funciona correctamente, teclea el programa en el ordenador y efectúa el mayor número de pruebas posibles, comprobando cuidadosamente que el resultado calculado es correcto. Presta especial atención a configuraciones extremas o singulares de los datos (los que pueden provocar divisiones por cero o valores muy grandes, o muy pequeños, o negativos, etc.). Si el programa calcula algo diferente de lo esperado o si se aborta la ejecución del programa por los errores detectados, vuelve al paso 2.

Pruebas unitarias

Correr una batería de pruebas *manualmente* cada vez que cambias algo de un programa, bien porque sigues una aproximación de refinamientos sucesivos, bien porque estás corrigiendo errores detectados tardíamente, es una labor ardua y aburridísima. Probablemente empezarás a encontrar todo tipo de excusas y justificaciones para no pasar tu batería de pruebas tras cada cambio.

Para evitar que la pereza pueda con las buenas prácticas, hay herramientas que permiten construir pruebas cuya ejecución es automática. Tras cada ejecución se muestra un informe con las pruebas que fallaron y la razón de que fallaran. Algunas de estas herramientas cuentan con una interfaz gráfica que muestra una luz verde si todas las pruebas se ejecutaron sin detectar fallo alguno, y roja si alguna falló.

La pruebas automáticas que testean pequeñas unidades funcionales, como los programas que estamos escribiendo por el momento, se denominan «pruebas unitarias» (del inglés «unit tests»).

Nadie es capaz de hacer un programa suficientemente largo de una sentada, empezando a escribir por la primera línea y acabando por la última, una tras otra, del mismo modo que nadie es capaz de escribir una novela o una sinfonía de una sentada[3]. Lo normal es empezar con un borrador e ir refinándolo, mejorándolo poco a poco.

Un error frecuente es tratar de diseñar el programa directamente sobre el ordenador, escribiéndolo a bote pronto. Es más, hay estudiantes que se atreven a empezar con la escritura de un programa sin haber entendido bien el enunciado del problema que se pretende resolver. Es fácil pillarlos en falta: no saben resolver a mano un caso particular del problema. Una buena práctica, pues, es solucionar manualmente unos pocos ejemplos concretos para estar seguros de que conocemos bien lo que se nos pide y cómo calcularlo. Una vez superada esta fase, estarás en condiciones de elaborar un borrador con los pasos que has de seguir. Créenos: es mejor que pienses un rato y diseñes un borrador del algoritmo sobre papel. Cuando estés muy seguro de la validez del algoritmo, impleméntalo en Python y pruébalo sobre el ordenador. Las pruebas con el ordenador te ayudarán a encontrar errores.

Ciertamente es posible utilizar el ordenador directamente, como si fuera el papel. Nada impide que el primer borrador lo hagas ya en pantalla, pero, si lo haces, verás que:

[3]Aunque hay excepciones: cuenta la leyenda que Mozart escribía sus obras de principio a fin, sin volver atrás para efectuar correcciones.

- Los detalles del lenguaje de programación interferirán en el diseño del algoritmo («¿he de poner dos puntos al final de la línea?», «¿uso marcas de formato para imprimir los resultados?», etc.): cuando piensas en el método de resolución del problema es mejor hacerlo con cierto grado de abstracción, sin tener en cuenta todas las particularidades de la notación.
- Si ya has tecleado un programa y sigue una aproximación incorrecta, te resultará más molesto prescindir de él que si no lo has tecleado aún. Esta molestia conduce a la tentación de ir poniendo parches a tu deficiente programa para ver si se puede arreglar algo. El resultado será, muy probablemente, un programa ilegible, pésimamente organizado... y erróneo. Te costará la mitad de tiempo empezar de cero, pero esta vez haciendo bien las cosas: pensando antes de escribir nada.

El síndrome «A mí nunca se me hubiera ocurrido esto»

Programar es una actividad que requiere un gran esfuerzo intelectual, no cabe duda, pero sobre todo, ahora que empiezas, es una actividad radicalmente diferente de cualquier otra para la que te vienes preparando desde la enseñanza primaria. Llevas muchos años aprendiendo lengua, matemáticas, física, etc., pero nunca antes habías programado. Los programas que hemos visto en este capítulo te deben parecer muy complicados, cuando no lo son tanto.

La reacción de muchos estudiantes al ver la solución que da el profesor o el libro de texto a un problema de programación es decirse «a mí nunca se me hubiera ocurrido esto». Debes tener en cuenta dos factores:

- La solución final muchas veces esconde la línea de razonamiento que permitió llegar a ese programa concreto. Nadie construye los programas de golpe: por regla general se hacen siguiendo refinamientos sucesivos a partir de una primera versión bastante tosca.
- La solución que se te presenta sigue la línea de razonamiento de una persona concreta: el profesor. Puede que tu línea de razonamiento sea diferente y, sin embargo, igualmente válida (¡o incluso mejor!), así que tu programa puede no parecerse en nada al suyo y, a la vez, ser correcto. No obstante, te conviene estudiar la solución que te propone el profesor: la lectura de programas escritos por otras personas es un buen método de aprendizaje y, probablemente, la solución que te ofrece resuelva cuestiones en las que no habías reparado (aunque solo sea porque el profesor lleva más años que tú en esto de programar).

4.1.7. Un nuevo refinamiento del programa de ejemplo

Parece que nuestro programa ya funciona correctamente. Probemos a resolver esta ecuación:

$$x^2 + 2x + 3 = 0$$

```
Programa para la resolución de la ecuación a x*x + b x + c= 0.
Valor de a: 1↵
Valor de b: 2↵
Valor de c: 3↵
Traceback (most recent call last):
  File "segundo_grado.py", line 10, in <module>
    x1 = (-b + sqrt(b**2 - 4*a*c)) / (2 * a)
ValueError: math domain error
```

¡Nuevamente un error! El mensaje de error es diferente de los anteriores y es un «error de dominio matemático».

El problema es que estamos intentando calcular la raíz cuadrada de un número negativo en la línea 10. El resultado es un número complejo, pero el módulo *math* no «sabe» de números complejos, así que *sqrt* falla y se produce un error. También en la línea 11 se tiene que calcular la raíz cuadrada de un número negativo, pero como la línea 10 se ejecuta en primer lugar, es ahí donde se produce el error y se aborta la ejecución. La línea 11 no llega a ejecutarse.

Podemos controlar este error asegurándonos de que el término b^2-4ac (que recibe el nombre de «discriminante») sea mayor o igual que cero *antes* de calcular la raíz cuadrada:

```
segundo_grado.py
from math import sqrt   # La función sqrt calcula la raíz cuadrada de un número.

print('Programa␣para␣la␣resolución␣de␣la␣ecuación␣a␣x*x␣+␣b␣x␣+␣c=␣0.')

a = float(input('Valor␣de␣a:␣'))
b = float(input('Valor␣de␣b:␣'))
c = float(input('Valor␣de␣c:␣'))

if a != 0:
    if b**2 - 4*a*c >= 0:
        x1 = (-b + sqrt(b**2 - 4*a*c)) / (2 * a)
        x2 = (-b - sqrt(b**2 - 4*a*c)) / (2 * a)
        print('Soluciones:␣x1={0:.3f}␣y␣x2={1:.3f}'.format(x1, x2))
    else:
        print('No␣hay␣soluciones␣reales.')
else:
    if b != 0:
        x = -c / b
        print('Solución:␣x={0:.3f}'.format(x))
    else:
        if c != 0:
            print('La␣ecuación␣no␣tiene␣solución.')
        else:
            print('La␣ecuación␣tiene␣infinitas␣soluciones.')
```

▶ 74 Un programador ha intentado solucionar el problema del discriminante negativo con un programa que empieza así:

```
segundo_grado.py
from math import sqrt

print('Programa␣para␣la␣resolución␣de␣la␣ecuación␣a␣x*x␣+␣b␣x␣+␣c=␣0.')

a = float(input('Valor␣de␣a:␣'))
b = float(input('Valor␣de␣b:␣'))
c = float(input('Valor␣de␣c:␣'))

if a != 0:
    if sqrt(b**2 - 4*a*c) >= 0:
        x1 = (-b + sqrt(b**2 - 4*a*c)) / (2 * a)
        x2 = (-b - sqrt(b**2 - 4*a*c)) / (2 * a)
        ...
```

Evidentemente, el programa es incorrecto y te sorprenderá saber que algunos estudiantes proponen soluciones similares a esta. El problema estriba en el posible valor negativo *del argumento* de *sqrt*, así que la comparación es incorrecta, pues pregunta por el signo *de la raíz* de dicho argumento. Pero el programa no llega siquiera a dar solución alguna (bien o mal calculada) cuando lo ejecutamos con, por ejemplo, $a = 4$, $b = 2$ y $c = 4$. ¿Qué sale por pantalla en ese caso? ¿Por qué?

Dado que solo hemos usado sentencias condicionales para controlar los errores, es posible que te hayas llevado la impresión de que esta es su única utilidad. En absoluto. Vamos a utilizar una sentencia condicional con otro propósito. Mira qué ocurre cuando tratamos de resolver la ecuación $x^2 - 2x + 1 = 0$:

```
Programa para la resolución de la ecuación a x*x + b x + c= 0.
Valor de a: 1↵
Valor de b: -2↵
Valor de c: 1↵
```

```
Soluciones: x1=1.000 y x2=1.000
```

Las dos soluciones son iguales, y queda un tanto extraño que el programa muestre el mismo valor dos veces. Hagamos que, cuando las dos soluciones sean iguales, solo se muestre una de ellas:

```
segundo_grado.py
from math import sqrt  # La función sqrt calcula la raíz cuadrada de un número.

print('Programa para la resolución de la ecuación a x*x + b x + c= 0.')

a = float(input('Valor de a: '))
b = float(input('Valor de b: '))
c = float(input('Valor de c: '))

if a != 0:
    if b**2 - 4*a*c >= 0:
        x1 = (-b + sqrt(b**2 - 4*a*c)) / (2 * a)
        x2 = (-b - sqrt(b**2 - 4*a*c)) / (2 * a)
        if x1 == x2:
            print('Solución: x={0:.3f}'.format(x1))
        else:
            print('Soluciones: x1={0:.3f} y x2={1:.3f}'.format(x1, x2))
    else:
        print('No hay soluciones reales.')
else:
    if b != 0:
        x = -c / b
        print('Solución: x={0:.3f}'.format(x))
    else:
        if c != 0:
            print('La ecuación no tiene solución.')
        else:
            print('La ecuación tiene infinitas soluciones.')
```

```
Programa para la resolución de la ecuación a x*x + b x + c= 0.
Valor de a: 1↵
Valor de b: -2↵
Valor de c: 1↵
Solución: x=1.000
```

4.1.8. Otro ejemplo: máximo de una serie de números

Ahora que sabemos utilizar sentencias condicionales, vamos con un problema sencillo, pero que es todo un clásico en el aprendizaje de la programación: el cálculo del máximo de una serie de números.

Empezaremos por pedirle al usuario dos números enteros y le mostraremos por pantalla cuál es el mayor de los dos.

Estudia esta solución, a ver qué te parece:

```
maximo.py
a = int(input('Dame el primer número: '))
b = int(input('Dame el segundo número: '))

if a > b:
    máximo = a
else:
    máximo = b

print('El máximo es', máximo)
```

Optimización

Podemos plantear un nuevo refinamiento que tiene por objeto hacer un programa más rápido, más eficiente. Fíjate en que en las líneas 10, 11 y 12 del último programa se calcula cada vez la expresión `b**2 - 4*a*c`. ¿Para qué hacer tres veces un mismo cálculo? Si las tres veces el resultado va a ser el mismo, ¿no es una pérdida de tiempo repetir el cálculo? Podemos efectuar una sola vez el cálculo y guardar el resultado en una variable.

```python
from math import sqrt   # La función sqrt calcula la raíz cuadrada de un número.

print('Programa para la resolución de la ecuación a x*x + b x + c = 0.')

a = float(input('Valor de a: '))
b = float(input('Valor de b: '))
c = float(input('Valor de c: '))

if a != 0:
    discriminante = b**2 - 4*a*c
    if discriminante >= 0:
        x1 = (-b + sqrt(discriminante)) / (2 * a)
        x2 = (-b - sqrt(discriminante)) / (2 * a)
        if x1 == x2:
            print('Solución: x={0:.3f}'.format(x1))
        else:
            print('Soluciones: x1={0:.3f} y x2={1:.3f}'.format(x1, x2))
    else:
        print('No hay soluciones reales.')
else:
    if b != 0:
        x = -c / b
        print('Solución: x={0:.3f}'.format(x))
    else:
        if c != 0:
            print('La ecuación no tiene solución.')
        else:
            print('La ecuación tiene infinitas soluciones.')
```

Hacer que un programa funcione más eficientemente es *optimizar* el programa. No te obsesiones ahora con la optimización: estás aprendiendo a programar. Asegúrate de que tus programas funcionan correctamente, que ya habrá tiempo para optimizar.

▶ 75 ¿Qué líneas del último programa se ejecutan y qué resultado aparece por pantalla en cada uno de estos casos?

1) $a = 2$ y $b = 3$.

2) $a = 3$ y $b = 2$.

3) $a = -2$ y $b = 0$.

4) $a = 1$ y $b = 1$.

Analiza con cuidado el último caso. Observa que los dos números son iguales. ¿Cuál es, pues, el máximo? ¿Es correcto el resultado del programa?

▶ 76 Un aprendiz de programador ha diseñado este otro programa para calcular el máximo de dos números:

maximo.py

```python
a = int(input('Dame el primer número: '))
b = int(input('Dame el segundo número: '))

if a > b:
    máximo = a
```

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

```
if b > a:
    máximo = b

print('El␣máximo␣es', máximo)
```

¿Es correcto? ¿Qué pasa si introducimos dos números iguales?

Vamos con un problema más complicado: el cálculo del máximo de tres números enteros (que llamaremos *a*, *b* y *c*). He aquí una estrategia posible:

1) Me pregunto si *a* es mayor que *b* y, si es así, de momento *a* es candidato a ser el mayor, pero no sé si lo será definitivamente hasta compararlo con *c*. Me pregunto, pues, si *a* es mayor que *c*.
    1) Si *a* también es mayor que *c*, está claro que *a* es el mayor de los tres.
    2) Y si no, *c* es el mayor de los tres.
2) Pero si no es así, es decir, si *a* es menor o igual que *b*, el número *b* es, de momento, mi candidato a número mayor. Falta compararlo con *c*.
    1) Si también es mayor que *c*, entonces *b* es el mayor.
    2) Y si no, entonces *c* es el mayor.

Ahora que hemos diseñado el procedimiento, construyamos un programa Python que implemente ese algoritmo:

maximo.py

```
1  a = int(input('Dame␣el␣primer␣número:␣'))
2  b = int(input('Dame␣el␣segundo␣número:␣'))
3  c = int(input('Dame␣el␣tercer␣número:␣'))
4
5  if a > b:
6      if a > c:
7          máximo = a
8      else:
9          máximo = c
10 else:
11     if b > c:
12         máximo = b
13     else:
14         máximo = c
15
16 print('El␣máximo␣es', máximo)
```

▶ 77 ¿Qué secuencia de líneas de este último programa se ejecutará en cada uno de estos casos?

1) $a = 2$, $b = 3$ y $c = 4$.
2) $a = 3$, $b = 2$ y $c = 4$.
3) $a = 1$, $b = 1$ y $c = 1$.

Puede que la solución que hemos propuesto te parezca extraña y que tú hayas diseñado un programa muy diferente. Es normal. No existe un único programa para solucionar un problema determinado y cada persona desarrolla un estilo propio en el diseño de los programas. Si el que se propone como solución no es igual al tuyo, el tuyo no tiene por qué ser erróneo; quizá solo sea distinto. Por ejemplo, este otro programa también calcula el máximo de tres números, y es muy diferente del que hemos propuesto antes:

maximo.py
```
a = int(input('Dame␣el␣primer␣número:␣'))
b = int(input('Dame␣el␣segundo␣número:␣'))
c = int(input('Dame␣el␣tercer␣número:␣'))

candidato = a
if b > candidato:
    candidato = b
if c > candidato:
    candidato = c
máximo = candidato

print('El␣máximo␣es', máximo)
```

▶ 78 Diseña un programa que calcule el máximo de 5 números enteros. Si sigues una estrategia similar a la de la primera solución propuesta para el problema del máximo de 3 números, tendrás problemas. Intenta resolverlo como en el último programa de ejemplo, es decir, con un «candidato a valor máximo» que se va actualizando al compararse con cada número.

▶ 79 Diseña un programa que calcule la menor de cinco palabras dadas; es decir, la primera palabra de las cinco en orden alfabético. Aceptaremos que las mayúsculas son «alfabéticamente» menores que las minúsculas, de acuerdo con la tabla ASCII.

▶ 80 Diseña un programa que calcule la menor de cinco palabras dadas; es decir, la primera palabra de las cinco en orden alfabético. *No* aceptaremos que las mayúsculas sean «alfabéticamente» menores que las minúsculas. O sea, `'pepita'` es menor que `'Pepito'`.

▶ 81 Diseña un programa que, dados cinco números enteros, determine cuál de los cuatro últimos números es más cercano al primero. (Por ejemplo, si el usuario introduce los números 2, 6, 4, 1 y 10, el programa responderá que el número más cercano al 2 es el 1).

▶ 82 Diseña un programa que, dados cinco puntos en el plano, determine cuál de los cuatro últimos puntos es más cercano al primero. Un punto se representará con dos variables: una para la abcisa y otra para la ordenada. La distancia entre dos puntos (x_1, y_1) y (x_2, y_2) es $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

Las condiciones pueden incluir cualquier expresión cuyo resultado sea interpretable en términos de cierto o falso. Podemos incluir, pues, expresiones lógicas tan complicadas como deseemos. Fíjate en el siguiente programa, que sigue una aproximación diferente para resolver el problema del cálculo del máximo de tres números:

maximo.py
```
a = int(input('Dame␣el␣primer␣número:␣'))
b = int(input('Dame␣el␣segundo␣número:␣'))
c = int(input('Dame␣el␣tercer␣número:␣'))

if a >= b and a >= c:
    máximo = a
if b >= a and b >= c:
    máximo = b
if c >= a and c >= b:
    máximo = c

print('El␣máximo␣es', máximo)
```

La expresión *a* >= *b* **and** *a* >= *c*, por ejemplo, se lee «*a* es mayor o igual que *b* y *a* es mayor o igual que *c*».

▶ 83 Indica en cada uno de los siguientes programas qué valores o rangos de valores provocan la aparición de los distintos mensajes:

1) aparcar.py

```
día = int(input('Dime␣qué␣día␣es␣hoy:␣'))

if 0 < día <= 15:
    print('Puedes␣aparcar␣en␣el␣lado␣izquierdo␣de␣la␣calle')
else:
    if día < 32:
        print('Puedes␣aparcar␣en␣el␣lado␣derecho␣de␣la␣calle')
    else:
        print('Ningún␣mes␣tiene␣{0}␣días.'.format(día))
```

2) estaciones.py

```
mes = int(input('Dame␣un␣mes:␣'))

if 1 <= mes <= 3:
    print('Invierno.')
else:
    if mes == 4 or mes == 5 or mes == 6:
        print('Primavera.')
    else:
        if not (mes < 7 or 9 < mes):
            print('Verano.')
        else:
            if not (mes != 10 and mes != 11 and mes != 12):
                print('Otoño.')
            else:
                print('Ningún␣año␣tiene␣{0}␣meses.'.format(mes))
```

3) identificador.py

```
car = input('Dame␣un␣carácter:␣')

if 'a' <= car.lower() <= 'z' or car == '_':
    print('Este␣carácter␣es␣válido␣en␣un␣identificador␣en␣Python2.')
else:
    if not (car < '0' or '9' < car):
        print('Un␣dígito␣es␣válido␣en␣un␣identificador␣en␣Python2,', end='')
        print('siempre␣que␣no␣sea␣el␣primer␣carácter.')
    else:
        print('Carácter␣no␣válido␣para␣formar␣un␣identificador␣en␣Python2.')
```

4) bisiesto.py

```
año = int(input('Dame␣un␣año:␣'))

if año % 4 == 0 and (año % 100 != 0 or año % 400 == 0):
    print('El␣año␣{0}␣es␣bisiesto.'.format(año))
else:
    print('El␣año␣{0}␣no␣es␣bisiesto.'.format(año))
```

▶ 84 La fórmula $C' = C \cdot (1 + x/100)^n$ nos permite obtener el capital final que lograremos a partir de un capital inicial (C), una tasa de interés anual (x) en tanto por cien y un número de años (n). Si lo que nos interesa conocer es el número de años n que tardaremos en lograr un capital final C' partiendo de un capital inicial C a una tasa de interés anual x, podemos despejar n en la fórmula de la siguiente manera:

$$n = \frac{\log(C') - \log(C)}{\log(1 + x/100)}$$

Diseña un programa Python que obtenga el número de años que se tarda en conseguir un capital final dado a partir de un capital inicial y una tasa de interés anual también dados. El programa debe tener en cuenta cuándo se puede realizar el cálculo y cuándo no en función del valor de la tasa de interés (para evitar una división por cero, el cálculo de logaritmos de valores negativos, etc.)... con una excepción: si C y C' son iguales, el número de años es 0 independientemente de la tasa de interés (incluso de la que provocaría una división por cero).

(Ejemplos: Para obtener 11,000 € con una inversión de 10,000 € al 5% anual es necesario esperar 1.9535 años. Obtener 11,000 € con una inversión de 10,000 € al 0% anual es imposible. Para obtener 10,000 € con una inversión de 10,000 € no hay que esperar nada, sea cual sea el interés).

▶ 85 Diseña un programa que, dado un número real que debe representar la calificación numérica de un examen, proporcione la calificación cualitativa correspondiente al número dado. La calificación cualitativa será una de las siguientes: «Suspenso» (nota menor que 5), «Aprobado» (nota mayor o igual que 5, pero menor que 7), «Notable» (nota mayor o igual que 7, pero menor que 9), «Sobresaliente» (nota mayor o igual que 9, pero menor que 10), «Matrícula de Honor» (nota 10).

▶ 86 Diseña un programa que, dado un carácter cualquiera, lo identifique como vocal minúscula, vocal mayúscula, consonante minúscula, consonante mayúscula u otro tipo de carácter. (Considera únicamente letras del alfabeto inglés).

4.1.9. Evaluación con cortocircuitos

La evaluación de expresiones lógicas tiene algo especial. Observa la condición de este **if**:

```
if a == 0 or 1/a > 1:
    ...
```

¿Puede provocar una división por cero? No, nunca. Observa que si *a* vale cero, el primer término del **or** es True. Como la evaluación de una *o lógica* de True con cualquier otro valor, True o False, es necesariamente True, Python *no evalúa* el segundo término y se ahorra así un esfuerzo innecesario.

Algo similar ocurre en este otro caso:

```
if a != 0 and 1/a > 1:
    ...
```

Si *a* es nulo, el valor de *a* **!=** 0 es falso, así que ya no se procede a evaluar la segunda parte de la expresión.

Al calcular el resultado de una expresión lógica, Python evalúa (siguiendo las reglas de asociatividad y precedencia oportunas) lo justo hasta conocer el resultado: cuando el primer término de un **or** es cierto, Python acaba y devuelve directamente cierto y cuando el primer término de un **and** es falso, Python acaba y devuelve directamente falso. Este modo de evaluación se conoce como *evaluación con cortocircuitos*.

▶ 87 ¿Por qué obtenemos un error en la siguiente sesión de trabajo con el intérprete interactivo?

```
>>> a = 0↵
>>> if 1/a > 1 and a != 0:↵
...     print(a)↵
... ↵
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ZeroDivisionError: division by zero
```

De Morgan

Las expresiones lógicas pueden resultar complicadas, pero es que los programas hacen, en ocasiones, comprobaciones complicadas. Tal vez las más difíciles de entender son las que comportan algún tipo de negación, pues generalmente nos resulta más difícil razonar en sentido negativo que afirmativo. A los que empiezan a programar les lían muy frecuentemente las negaciones combinadas con **or** o **and**. Veamos algún ejemplo «de juguete». Supón que para aprobar una asignatura hay que obtener más de un 5 en dos exámenes parciales, y que la nota de cada uno de ellos está disponible en las variables *parcial1* y *parcial2*, respectivamente. Estas líneas de programa muestran el mensaje «Has suspendido.» cuando no has obtenido al menos un 5 en los dos exámenes:

```
if not (parcial1 >= 5.0 and parcial2 >= 5.0):
    print('Has suspendido.')
```

Lee bien la condición: «si no es cierto que has sacado al menos un 5 *en ambos* (por eso el **and**) parciales. . . ». Ahora fíjate en este otro fragmento:

```
if not parcial1 >= 5.0 or not parcial2 >= 5.0:
  print('Has suspendido.')
```

Leámoslo: «si no has sacado al menos un cinco *en uno u otro* (por eso el **or**) parcial. . . ». O sea, los dos fragmentos son equivalentes: uno usa un **not** que se aplica al resultado de una operación **and**; el otro usa dos operadores **not** cuyos resultados se combinan con un operador **or**. Y sin embargo, dicen la misma cosa. Los lógicos utilizan una notación especial para representar esta equivalencia:

$$\neg(p \wedge q) \quad \longleftrightarrow \quad \neg p \vee \neg q,$$
$$\neg(p \vee q) \quad \longleftrightarrow \quad \neg p \wedge \neg q.$$

(Los lógicos usan '¬' para **not**, '∧' para **and** y '∨' para **or**). Estas relaciones se deben al matemático De Morgan, y por su nombre se las conoce. Si es la primera vez que las ves, te resultarán chocantes, pero si piensas un poco, verás que son de sentido común.

Hemos observado que los estudiantes cometéis errores cuando hay que expresar la condición contraria a una como «*a* **and** *b*». Muchos escribís «**not** *a* **and** **not** *b*» y está mal. La negación correcta sería «**not** (*a* **and** *b*)» o, por De Morgan, «**not** *a* **or** **not** *b*». ¿Cuál sería, por cierto, la negación de «*a* **or** **not** *b*»?

4.1.10. Un último problema: menús de usuario

Ya casi acabamos esta (larguísima) sección. Introduciremos una nueva estructura sintáctica planteando un nuevo problema. El problema es el siguiente: imagina que tenemos un programa que a partir del radio de una circunferencia calcula su diámetro, perímetro o área. Solo queremos mostrar al usuario una de las tres cosas, el diámetro, el perímetro o el área; la que él desee, pero solo una.

Nuestro programa podría empezar pidiendo el radio del círculo. A continuación, podría mostrar un *menú* con tres *opciones*: «calcular el diámetro», «calcular el perímetro» y «calcular el área». Podríamos etiquetar cada opción con una letra y hacer que el usuario tecleara una de ellas. En función de la letra tecleada, calcularíamos una cosa u otra.

Analiza este programa:

circulo.py
```
from math import pi

radio = float(input('Dame el radio de un círculo: '))

# Menú
print('Escoge una opción: ')
print('a) Calcular el diámetro.')
print('b) Calcular el perímetro.')
print('c) Calcular el área.')
```

```
opción = input('Teclea␣a,␣b␣o␣c␣y␣pulsa␣el␣retorno␣de␣carro:␣')

if opción == 'a':  # Cálculo del diámetro.
    diámetro = 2 * radio
    print('El␣diámetro␣es␣{0}.'.format(diámetro))
else:
    if opción == 'b': # Cálculo del perímetro.
        perímetro = 2 * pi * radio
        print('El␣perímetro␣es␣{0}.'.format(perímetro))
    else:
        if opción == 'c': # Cálculo del área.
            área = pi * radio ** 2
            print('El␣área␣es␣{0}.'.format(área))
```

Ejecutemos el programa y seleccionemos la segunda opción:

```
Dame el radio de un círculo: 3↵
Escoge una opción:
a) Calcular el diámetro.
b) Calcular el perímetro.
c) Calcular el área.
Teclea a, b o c y pulsa el retorno de carro: b↵
El perímetro es 18.84955592153876.
```

Ejecutémoslo de nuevo, pero seleccionando esta vez la tercera opción:

```
Dame el radio de un círculo: 3↵
Escoge una opción:
a) Calcular el diámetro.
b) Calcular el perímetro.
c) Calcular el área.
Teclea a, b o c y pulsa el retorno de carro: c↵
El área es 28.274333882308138.
```

▶ 88 Nuestro aprendiz de programador ha tecleado en su ordenador el último programa, pero se ha despistado y ha escrito esto:

circulo.py

```
from math import pi

radio = float(input('Dame␣el␣radio␣de␣un␣círculo:␣'))

# Menú
print('Escoge␣una␣opción:␣')
print('a)␣Calcular␣el␣diámetro.')
print('b)␣Calcular␣el␣perímetro.')
print('c)␣Calcular␣el␣área.')
opción = input('Teclea␣a,␣b␣o␣c␣y␣pulsa␣el␣retorno␣de␣carro:␣')

if opción == a:   # Cálculo del diámetro.
    diámetro = 2 * radio
    print('El␣diámetro␣es␣{0}.'.format(diámetro))
else:
    if opción == b: # Cálculo del perímetro.
        perímetro = 2 * pi * radio
        print('El␣perímetro␣es␣{0}.'.format(perímetro))
    else:
        if opción == c: # Cálculo del área.
            área = pi * radio ** 2
            print('El␣área␣es␣{0}.'.format(área))
```

Las líneas sombreadas son diferentes de sus equivalentes del programa original. ¿Funcionará el programa del aprendiz? Si no es así, ¿por qué motivo?

Acabemos de pulir nuestro programa. Cuando el usuario no escribe ni la *a*, ni la *b*, ni la *c* al tratar de seleccionar una de las opciones, deberíamos decirle que se ha equivocado:

```
circulo.py
1 from math import pi
2
3 radio = float(input('Dame el radio de un círculo: '))
4
5 # Menú
6 print('Escoge una opción: ')
7 print('a) Calcular el diámetro.')
8 print('b) Calcular el perímetro.')
9 print('c) Calcular el área.')
10 opción = input('Teclea a, b o c y pulsa el retorno de carro: ')
11
12 if opción == 'a':  # Cálculo del diámetro.
13     diámetro = 2 * radio
14     print('El diámetro es {0}.'.format(diámetro))
15 else:
16     if opción == 'b': # Cálculo del perímetro.
17         perímetro = 2 * pi * radio
18         print('El perímetro es {0}.'.format(perímetro))
19     else:
20         if opción == 'c': # Cálculo del área.
21             área = pi * radio ** 2
22             print('El área es {0}.'.format(área))
23         else:
24             print('Solo hay tres opciones: a, b o c.')
25             print('Tú has tecleado "{0}".'.format(opción))
```

▶ 89 Haz una traza del programa suponiendo que el usuario teclea la letra `d` cuando se le solicita una opción. ¿Qué líneas del programa se ejecutan?

▶ 90 El programa presenta un punto débil: si el usuario escribe una letra mayúscula en lugar de minúscula, no se selecciona ninguna opción. Modifica el programa para que también acepte letras mayúsculas.

4.1.11. Una forma compacta para estructuras condicionales múltiples (elif)

El último programa presenta un problema estético: la serie de líneas que permiten seleccionar el cálculo que hay que efectuar según la opción de menú seleccionada (líneas 12–25) parece más complicada de lo que realmente es. Cada opción aparece sangrada más a la derecha que la anterior, así que el cálculo del área acaba con tres niveles de sangrado. Imagina qué pasaría si el menú tuviera 8 o 9 opciones: ¡el programa acabaría tan a la derecha que prácticamente se saldría del papel! Python permite una forma compacta de expresar fragmentos de código de la siguiente forma:

```
if condición:
  ...
else:
  if otra condición:
    ...
```

Un **else** inmediatamente seguido por un **if** puede escribirse así:

```
if condición:
  ...
elif otra condición:
  ...
```

con lo que nos ahorramos un sangrado. El último programa se convertiría, pues, en este otro:

```
circulo.py
from math import pi

radio = float(input('Dame␣el␣radio␣de␣un␣círculo:␣'))

# Menú
print('Escoge␣una␣opción:␣')
print('a)␣Calcular␣el␣diámetro.')
print('b)␣Calcular␣el␣perímetro.')
print('c)␣Calcular␣el␣área.')
opción = input('Teclea␣a,␣b␣o␣c␣y␣pulsa␣el␣retorno␣de␣carro:␣')

if opción == 'a':  # Cálculo del diámetro.
    diámetro = 2 * radio
    print('El␣diámetro␣es␣{0}.'.format(diámetro))
elif opción == 'b': # Cálculo del perímetro.
    perímetro = 2 * pi * radio
    print('El␣perímetro␣es␣{0}.'.format(perímetro))
elif opción == 'c': # Cálculo del área.
    área = pi * radio ** 2
    print('El␣área␣es␣{0}.'.format(área))
else:
    print('Solo␣hay␣tres␣opciones:␣a,␣b␣o␣c.')
    print('Tú␣has␣tecleado␣"{0}".'.format(opción))
```

El programa es absolutamente equivalente, ocupa menos líneas y gana mucho en legibilidad: no solo evitamos mayores niveles de sangrado, también expresamos de forma clara que, en el fondo, todas esas condiciones están relacionadas.

Formas compactas: ¿complicando las cosas?

Puede que comprender la estructura condicional if te haya supuesto un esfuerzo considerable. A eso has tenido que añadir la forma if-else. ¡Y ahora el if-elif! Parece que no hacemos más que complicar las cosas. Más bien todo lo contrario: las formas if-else e if-elif (que también acepta un if-elif-else) debes considerarlas una ayuda. En realidad, ninguna de estas formas permite hacer cosas que no pudiéramos hacer con solo el if, aunque, eso sí, necesitando un esfuerzo mayor.

Mientras estés dando tus primeros pasos en la programación, si dudas sobre qué forma utilizar, trata de expresar tu idea con solo el if. Una vez tengas una solución, plantéate si tu programa se beneficiaría del uso de una forma compacta. Si es así, úsala. Más adelante seleccionarás instintivamente la forma más apropiada para cada caso. Bueno, eso cuando hayas adquirido bastante experiencia, y *solo* la adquirirás practicando.

▶ 91 Modifica la solución del ejercicio 85 usando ahora la estructura elif. ¿No te parece más legible la nueva solución?

4.2. Sentencias iterativas

Aún vamos a presentar una última reflexión sobre el programa de los menús. Cuando el usuario no escoge correctamente una opción del menú el programa le avisa, pero finaliza inmediatamente. Lo ideal sería que cuando el usuario se equivocara, el programa le pidiera de nuevo

una opción. Para eso sería necesario *repetir* la ejecución de las líneas 10–23. Una aproximación naïf consistiría, básicamente, en añadir al final una copia de esas líneas precedidas de un if que comprobara que el usuario se equivocó. Pero esa aproximación es muy mala: ¿qué pasaría si el usuario se equivocara una segunda vez? Cuando decimos que queremos *repetir* un fragmento del programa no nos referimos a *copiarlo* de nuevo, sino a *ejecutarlo* otra vez. Pero, ¿es posible expresar en este lenguaje que queremos que se repita la ejecución de un trozo del programa?

Python permite indicar que deseamos que se repita un trozo de programa de dos formas distintas: mediante la sentencia while y mediante la sentencia for. La primera de ellas es más general, por lo que la estudiaremos en primer lugar.

4.2.1. La sentencia while

En inglés, «while» significa «mientras». La sentencia while se usa así:

```
while condición:
    acción
    acción
    ...
    acción
```

y permite expresar en Python acciones cuyo significado es:

«*Mientras* se cumpla esta condición, repite estas acciones».

Las sentencias que denotan repetición se denominan *bucles*.

Vamos a empezar estudiando un ejemplo y viendo qué ocurre paso a paso. Estudia detenidamente este programa:

```
contador.py
1 i = 0
2 while i < 3:
3     print(i)
4     i += 1
5 print('Hecho')
```

Observa que la línea 2 finaliza con dos puntos (:) y que el sangrado indica que las líneas 3 y 4 dependen de la línea 2, pero no la línea 5. Podemos leer el programa así: primero, asigna a *i* el valor 0; a continuación, *mientras i* sea menor que 3, *repite estas acciones*: muestra por pantalla el valor de *i* e incrementa *i* en una unidad; finalmente, muestra por pantalla la palabra «Hecho».

Si ejecutamos el programa, por pantalla aparecerá el siguiente texto:

```
0
1
2
Hecho
```

Veamos qué ha ocurrido paso a paso con una traza.

- Se ha ejecutado la línea 1, con lo que *i* vale 0.

```
i = 0
while i < 3:
    print(i)
    i += 1
print('Hecho')
```

- Después, se ha ejecutado la línea 2, que dice «mientras *i* sea menor que 3, hacer…». Primero se ha evaluado la condición *i* < 3, que ha resultado ser cierta. Como la condición se satisface, deben ejecutarse las acciones supeditadas a esta línea (las dos siguientes, que están más sangradas).

```
i = 0
while i < 3:
    print(i)
    i += 1
print('Hecho')
```

- Se ejecuta en primer lugar la línea 3, que muestra el valor de *i* por pantalla. Aparece, pues, un cero.

```
i = 0
while i < 3:
    print(i)
    i += 1
print('Hecho')
```

- Se ejecuta a continuación la línea 4, que incrementa el valor de *i*. Ahora *i* vale 1.

```
i = 0
while i < 3:
    print(i)
    i += 1
print('Hecho')
```

- ¡Ojo!, ahora *no* pasamos a la línea 5, sino que volvemos a la línea 2. Cada vez que finalizamos la ejecución de las acciones que dependen de un while, volvemos a la línea del while.

```
i = 0
while i < 3:
    print(i)
    i += 1
print('Hecho')
```

- Estamos nuevamente en la línea 2, así que comprobamos si *i* es menor que 3. Es así, por lo que toca ejecutar de nuevo las líneas 3 y 4.

```
i = 0
while i < 3:
    print(i)
    i += 1
print('Hecho')
```

- Volvemos a ejecutar la línea 3, así que aparece un 1 por pantalla.

```
i = 0
while i < 3:
    print(i)
    i += 1
print('Hecho')
```

- Volvemos a ejecutar la línea 4, con lo que *i* vuelve a incrementarse y pasa de valer 1 a valer 2.

```
i = 0
while i < 3:
    print(i)
    i += 1
print('Hecho')
```

- Nuevamente pasamos a la línea 2. *Siempre* que acaba de ejecutarse la última acción de un bucle while, volvemos a la línea que contiene la palabra while. Como i sigue siendo menor que 3, deberemos repetir las acciones expresadas en las líneas 3 y 4.

```
i = 0
while i < 3:
    print(i)
    i += 1
print('Hecho')
```

- Así que ejecutamos otra vez la línea 3 y en pantalla aparece el número 2.

```
i = 0
while i < 3:
    print(i)
    i += 1
print('Hecho')
```

- Incrementamos de nuevo el valor de i, como indica la línea 4, así que i pasa de valer 2 a valer 3.

```
i = 0
while i < 3:
    print(i)
    i += 1
print('Hecho')
```

- Y de nuevo pasamos a la línea 2. Pero ahora ocurre algo especial: la condición no se satisface, pues i ya no es menor que 3. Como la condición ya no se satisface, no hay que ejecutar otra vez las líneas 3 y 4. Ahora hemos de ir a la línea 5, que es la primera línea que no está «dentro» del bucle.

```
i = 0
while i < 3:
    print(i)
    i += 1
print('Hecho')
```

- Se ejecuta la línea 5, que muestra por pantalla la palabra «Hecho» y finaliza el programa.

```
i = 0
while i < 3:
    print(i)
    i += 1
print('Hecho')
```

Pero, ¿por qué tanta complicación? Este otro programa muestra por pantalla lo mismo, se entiende más fácilmente y es más corto.

contador_simple.py
```
print(0)
print(1)
print(2)
print('Hecho')
```

Bueno, `contador.py` es un programa que solo pretende ilustrar el concepto de bucle, así que ciertamente no hace nada demasiado útil, pero aun así nos permite vislumbrar la potencia del concepto de iteración o repetición. Piensa en qué ocurre si modificamos un solo número del programa:

contador.py
```
i = 0
while i < 1000:
    print(i)
    i += 1
print('Hecho')
```

¿Puedes escribir fácilmente un programa que haga lo mismo y que no utilice bucles?

▶ 92 Haz una traza de este programa:

ejercicio_bucle.py
```
i = 0
while i <= 3:
    print(i)
    i += 1
print('Hecho')
```

▶ 93 Haz una traza de este programa:

ejercicio_bucle.py
```
i = 0
while i < 10:
    print(i)
    i += 2
print('Hecho')
```

▶ 94 Haz una traza de este programa:

ejercicio_bucle.py
```
i = 3
while i < 10:
    i += 2
    print(i)
print('Hecho')
```

▶ 95 Haz una traza de este programa:

ejercicio_bucle.py
```
i = 1
while i < 100:
    i *= 2
    print(i)
```

▶ 96 Haz una traza de este programa:

```
ejercicio_bucle.py
i = 10
while i < 2:
    i *= 2
    print(i)
```

▶ 97 Haz unas cuantas trazas de este programa para diferentes valores de *i*.

```
ejercicio_bucle.py
i = int(input('Valor␣inicial:␣'))
while i < 10:
    print(i)
    i += 1
```

¿Qué ocurre si el valor de *i* es mayor o igual que 10? ¿Y si es negativo?

▶ 98 Haz unas cuantas trazas de este programa para diferentes valores de *i* y de *límite*.

```
ejercicio_bucle.py
i = int(input('Valor␣inicial:␣'))
límite = int(input('Límite:␣'))
while i < límite:
    print(i)
    i += 1
```

▶ 99 Haz unas cuantas trazas de este programa para diferentes valores de *i*, de *límite* y de *incremento*.

```
ejercicio_bucle.py
i = int(input('Valor␣inicial:␣'))
límite = int(input('Límite:␣'))
incremento = int(input('Incremento:␣'))
while i < límite:
    print(i)
    i += incremento
```

▶ 100 Implementa un programa que muestre todos los múltiplos de 6 entre 6 y 150, ambos inclusive.

▶ 101 Implementa un programa que muestre todos los múltiplos de n entre n y $m \cdot n$, ambos inclusive, donde n y m son números introducidos por el usuario.

▶ 102 Implementa un programa que muestre todos los números potencia de 2 entre 2^0 y 2^{30}, ambos inclusive.

Bucles sin fin

Los bucles son muy útiles a la hora de confeccionar programas, pero también son peligrosos si no andas con cuidado: es posible que no finalicen nunca. Estudia este programa y verás qué queremos decir:

```
bucle_infinito.py
i = 0
while i < 10:
  print(i)
```

La condición del bucle siempre se satisface: dentro del bucle nunca se modifica el valor de *i*, y si *i* no se modifica, jamás llegará a valer 10 o más. El ordenador empieza a mostrar el número 0 una y otra vez, sin finalizar nunca. Es lo que denominamos un *bucle sin fin* o *bucle infinito*.

Cuando se ejecuta un bucle sin fin, el ordenador se queda como «colgado» y nunca nos devuelve el control. Si estás ejecutando un programa desde la línea de órdenes Unix o una consola de Windows, puedes abortarlo pulsando `C-c`. Si la ejecución tiene lugar en Eclipse/Pydev puedes abortar la ejecución del programa pulsando en el cuadrado rojo que aparece en la barra superior de la consola.

4.2.2. Un problema de ejemplo: cálculo de sumatorios

Ahora que ya hemos presentado lo fundamental de los bucles, vamos a resolver algunos problemas concretos. Empezaremos por un programa que calcula la suma de los 1000 primeros números, es decir, un programa que calcula el sumatorio

$$\sum_{i=1}^{1000} i,$$

o, lo que es lo mismo, el resultado de $1 + 2 + 3 + \cdots + 999 + 1000$.

Vamos paso a paso. La primera idea que suele venir a quienes aprenden a programar es reproducir la fórmula con una sola expresión Python, es decir:

```
sumatorio.py
sumatorio = 1 + 2 + 3 + ... + 999 + 1000
print(sumatorio)
```

Pero, obviamente, no funciona: los puntos suspensivos no significan nada para Python. Aunque una persona puede aplicar su intuición para deducir qué significan los puntos suspensivos en ese contexto, Python carece de intuición alguna: exige que todo se describa de forma precisa y rigurosa. Esa es la mayor dificultad de la programación: el nivel de detalle y precisión con el que hay que describir qué se quiere hacer.

Bien. Abordémoslo de otro modo. Vamos a intentar calcular el valor del sumatorio «acumulando» el valor de cada número en una variable. Analiza este otro programa (incompleto):

```
1    sumatorio = 0
2    sumatorio += 1
3    sumatorio += 2
4    sumatorio += 3
     ...
1000 sumatorio += 999
1001 sumatorio += 1000
1002 print(sumatorio)
```

Como programa no es el colmo de la elegancia. Fíjate en que, además, presenta una estructura casi repetitiva: las líneas de la 2 a la 1001 son todas de la forma

$$sumatorio \mathrel{+}= número$$

donde *número* va tomando todos los valores entre 1 y 1000. Ya que esa sentencia, con ligeras variaciones, se repite una y otra vez, vamos a tratar de utilizar un bucle. Empecemos construyendo un borrador incompleto que iremos refinando progresivamente:

sumatorio.py
```
sumatorio = 0
while condición:
    sumatorio += número
print(sumatorio)
```

Hemos dicho que *número* ha de tomar todos los valores crecientes desde 1 hasta 1000. Podemos usar una variable que, una vez inicializada, vaya tomando valores sucesivos con cada iteración del bucle:

sumatorio.py
```
sumatorio = 0
i = 1
while condición:
    sumatorio += i
    i += 1
print(sumatorio)
```

Solo resta indicar la condición con la que se decide si hemos de iterar de nuevo o, por el contrario, hemos de finalizar el bucle:

sumatorio.py
```
sumatorio = 0
i = 1
while i <= 1000:
    sumatorio += i
    i += 1
print(sumatorio)
```

▶ 103 Estudia las diferencias entre el siguiente programa y el último que hemos estudiado. ¿Producen ambos el mismo resultado?

sumatorio.py
```
sumatorio = 0
i = 0
while i < 1000:
    i += 1
    sumatorio += i
print(sumatorio)
```

▶ 104 Diseña un programa que calcule

$$\sum_{i=n}^{m} i,$$

donde n y m son números enteros que deberá introducir el usuario por teclado.

▶ 105 Modifica el programa anterior para que si $n > m$, el programa no efectúe ningún cálculo y muestre por pantalla un mensaje que diga que n debe ser menor o igual que m.

▶ 106 Queremos hacer un programa que calcule el factorial de un número entero positivo. El factorial de n se denota con $n!$, pero no existe ningún operador Python que permita efectuar este cálculo directamente. Sabiendo que

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$$

y que $0! = 1$, haz un programa que pida el valor de n y muestre por pantalla el resultado de calcular $n!$.

▶ 107 El número de combinaciones que podemos formar tomando m elementos de un conjunto con n elementos es:

$$C_m^n = \binom{n}{m} = \frac{n!}{(n-m)!\,m!}.$$

Diseña un programa que pida el valor de n y m y calcule C_m^n. (Ten en cuenta que n ha de ser mayor o igual que m).

(Puedes comprobar la validez de tu programa introduciendo los valores $n = 15$ y $m = 10$: el resultado es 3003).

4.2.3. Otro programa de ejemplo: requisitos en la entrada

Vamos con otro programa sencillo pero ilustrativo. Estudia este programa:

raiz.py
```
1 from math import sqrt
2
3 x = float(input('Introduce␣un␣número␣positivo:␣'))
4
5 print('La␣raíz␣cuadrada␣de␣{0}␣es␣{1}'.format(x, sqrt(x)))
```

Como puedes ver, es muy sencillo: pide un número (flotante) y muestra por pantalla su raíz cuadrada. Como *sqrt* no puede trabajar con números negativos, *pedimos* al usuario que introduzca un número positivo. Pero nada *obliga* al usuario a introducir un número positivo.

En lugar de adoptar una solución como las estudiadas anteriormente, esto es, evitando ejecutar el cálculo de la raíz cuadrada cuando el número es negativo con la ayuda de una sentencia condicional, vamos a obligar a que el usuario introduzca un número positivo *repitiendo* la sentencia de la línea 3 cuantas veces sea preciso. Dado que vamos a repetir un fragmento de programa, utilizaremos una sentencia **while**. En principio, nuestro programa presentará este aspecto:

raiz.py
```
from math import sqrt

while condición:
    x = float(input('Introduce␣un␣número␣positivo:␣'))

print('La␣raíz␣cuadrada␣de␣{0}␣es␣{1}'.format(x, sqrt(x)))
```

¿Qué condición poner? Está claro: el bucle debería leerse así «mientras x sea un valor inválido, hacer…», es decir, «mientras x sea menor que cero, hacer…»; y esa última frase se traduce a Python así:

raiz.py
```
from math import sqrt

while x < 0:
    x = float(input('Introduce␣un␣número␣positivo:␣'))

print('La␣raíz␣cuadrada␣de␣{0}␣es␣{1}'.format(x, sqrt(x)))
```

Pero el programa no funciona correctamente. Mira qué obtenemos al ejecutarlo:

```
Traceback (most recent call last):
  File "raiz.py", line 3, in <module>
    while x < 0:
NameError: name 'x' is not defined
```

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

Python nos indica que la variable *x* no está definida (no existe) en la línea 3. ¿Qué ocurre? Vayamos paso a paso: Python empieza ejecutando la línea 1, con lo que importa la función *sqrt* del módulo *math*; la línea 2 está en blanco, así que, a continuación, Python ejecuta la línea 3, lo cual pasa por saber si la condición del **while** es cierta o falsa. Y ahí se produce el error, pues se intenta conocer el valor de *x* cuando *x* no está inicializada. Es necesario, pues, inicializar antes la variable; pero, ¿con qué valor? Desde luego, no con un valor positivo. Si *x* empieza tomando un valor positivo, la línea 4 no se ejecutará. Probemos, por ejemplo, con el valor −1.

raiz.py
```
1 from math import sqrt
2
3 x = -1
4 while x < 0:
5     x = float(input('Introduce␣un␣número␣positivo:␣'))
6
7 print('La␣raíz␣cuadrada␣de␣{0}␣es␣{1}'.format(x, sqrt(x)))
```

Ahora sí. Hagamos una traza.

1) Empezamos ejecutando la línea 1, con lo que importa la función *sqrt*.

2) La línea 2 se ignora.

3) Ahora ejecutamos la línea 3, con lo que *x* vale −1.

4) En la línea 4 nos preguntamos: ¿es *x* menor que cero? La respuesta es sí, de modo que debemos ejecutar la línea 5.

5) La línea 5 hace que se solicite al usuario un valor para *x*. Supongamos que el usuario introduce un número negativo, por ejemplo, −3.

6) Como hemos llegado al final de un bucle **while**, volvemos a la línea 4 y nos volvemos a preguntar ¿es *x* menor que cero? De nuevo, la respuesta es sí, así que pasamos a la línea 5.

7) Supongamos que ahora el usuario introduce un número positivo, pongamos que el 16.

8) Por llegar al final de un bucle, toca volver a la línea 4 y plantearse la condición: ¿es *x* menor que cero? En este caso la respuesta es no, así que salimos del bucle y pasamos a ejecutar la línea 7, pues la línea 6 está vacía.

9) La línea 7 muestra por pantalla «`La raíz cuadrada de 16.000000 es 4.000000`». Y ya hemos acabado.

Fíjate en que las líneas 4–5 se pueden repetir cuantas veces haga falta: solo es posible salir del bucle introduciendo un valor positivo en *x*. Ciertamente hemos conseguido obligar al usuario a que los datos que introduce satisfagan una cierta restricción.

▶ 108 ¿Qué te parece esta otra versión del mismo programa?

raiz.py
```
1 from math import sqrt
2
3 x = float(input('Introduce␣un␣número␣positivo:␣'))
4 while x < 0:
5   x = float(input('Introduce␣un␣número␣positivo:␣'))
6
7 print('La␣raíz␣cuadrada␣de␣{0}␣es␣{1}'.format(x, sqrt(x)))
```

▶ 109 Diseña un programa que solicite la lectura de un número entre 0 y 10 (ambos inclusive). Si el usuario teclea un número fuera del rango válido, el programa solicitará nuevamente la introducción del valor cuantas veces sea menester.

▶ 110 Diseña un programa que solicite la lectura de un texto que no contenga letras mayúsculas. Si el usuario teclea una letra mayúscula, el programa solicitará nuevamente la introducción del texto cuantas veces sea preciso.

▶ 111 Haz un programa que vaya leyendo números y mostrándolos por pantalla hasta que el usuario introduzca un número negativo. En ese momento, el programa mostrará un mensaje de despedida y finalizará su ejecución.

▶ 112 Haz un programa que vaya leyendo números hasta que el usuario introduzca un número negativo. En ese momento, el programa mostrará por pantalla el número mayor de cuantos ha visto.

4.2.4. Mejorando el programa de los menús

Al acabar la sección dedicada a sentencias condicionales presentamos este programa:

circulo.py
```
1  from math import pi
2
3  radio = float(input('Dame␣el␣radio␣de␣un␣círculo:␣'))
4
5  print('Escoge␣una␣opción:␣')
6  print('a)␣Calcular␣el␣diámetro.')
7  print('b)␣Calcular␣el␣perímetro.')
8  print('c)␣Calcular␣el␣área.')
9  opción = input('Teclea␣a,␣b␣o␣c␣y␣pulsa␣el␣retorno␣de␣carro:␣')
10
11 if opción == 'a':
12     diámetro = 2 * radio
13     print('El␣diámetro␣es', diámetro)
14 elif opción == 'b':
15     perímetro = 2 * pi * radio
16     print('El␣perímetro␣es', perímetro)
17 elif opción == 'c':
18     área = pi * radio ** 2
19     print('El␣área␣es', área)
20 else:
21     print('Solo␣hay␣tres␣opciones:␣a,␣b␣o␣c.␣Tú␣has␣tecleado', opción)
```

Y al empezar esta sección, dijimos que cuando el usuario no introduce correctamente una de las tres opciones del menú nos gustaría volver a mostrar el menú hasta que escoja una opción válida.

En principio, si queremos que el menú vuelva a aparecer por pantalla cuando el usuario se equivoca, deberemos repetir desde la línea 5 hasta la última, así que la sentencia **while** deberá aparecer inmediatamente antes de la quinta línea. El borrador del programa puede quedar así:

ϟ circulo.py
```
1  from math import pi
2
3  radio = float(input('Dame␣el␣radio␣de␣un␣círculo:␣'))
4
5  while opción < 'a' or opción > 'c':
6      print('Escoge␣una␣opción:␣')
7      print('a)␣Calcular␣el␣diámetro.')
8      print('b)␣Calcular␣el␣perímetro.')
9      print('c)␣Calcular␣el␣área.')
10     opción = input('Teclea␣a,␣b␣o␣c␣y␣pulsa␣el␣retorno␣de␣carro:␣')
11     if opción == 'a':
12         diámetro = 2 * radio
```

```
13         print('El diámetro es', diámetro)
14     elif opción == 'b':
15         perímetro = 2 * pi * radio
16         print('El perímetro es', perímetro)
17     elif opción == 'c':
18         área = pi * radio ** 2
19         print('El área es', área)
20     else:
21         print('Solo hay tres opciones: a, b o c. Tú has tecleado', opción)
```

Parece correcto, pero no lo es. ¿Por qué? El error estriba en que *opción* no existe la primera vez que ejecutamos la línea 5. ¡Nos hemos olvidado de inicializar la variable *opción*! Desde luego, el valor inicial de *opción* no debería ser `'a'`, `'b'` o `'c'`, pues entonces el bucle no se ejecutaría (piensa por qué). Cualquier otro valor hará que el programa funcione. Nosotros utilizaremos la cadena vacía para inicializar *opción*:

```
circulo.py
1  from math import pi
2
3  radio = float(input('Dame el radio de un círculo: '))
4
5  opción = ''
6  while opción < 'a' or opción > 'c':
7      print('Escoge una opción: ')
8      print('a) Calcular el diámetro.')
9      print('b) Calcular el perímetro.')
10     print('c) Calcular el área.')
11     opción = input('Teclea a, b o c y pulsa el retorno de carro: ')
12     if opción == 'a':
13         diámetro = 2 * radio
14         print('El diámetro es', diámetro)
15     elif opción == 'b':
16         perímetro = 2 * pi * radio
17         print('El perímetro es', perímetro)
18     elif opción == 'c':
19         área = pi * radio ** 2
20         print('El área es', área)
21     else:
22         print('Solo hay tres opciones: a, b o c. Tú has tecleado', opción)
```

▶ 113 ¿Es correcto este otro programa? ¿En qué se diferencia del anterior? ¿Cuál te parece mejor (si es que alguno de ellos te parece mejor)?

```
circulo.py
1  from math import pi
2
3  radio = float(input('Dame el radio de un círculo: '))
4
5  opción = ''
6  while opción < 'a' or opción > 'c':
7      print('Escoge una opción: ')
8      print('a) Calcular el diámetro.')
9      print('b) Calcular el perímetro.')
10     print('c) Calcular el área.')
11     opción = input('Teclea a, b o c y pulsa el retorno de carro: ')
12     if opción < 'a' or opción > 'c':
13         print('Solo hay tres opciones: a, b o c. Tú has tecleado', opción)
14
15 if opción == 'a':
16     diámetro = 2 * radio
```

```
    print('El␣diámetro␣es', diámetro)
elif opción == 'b':
    perímetro = 2 * pi * radio
    print('El␣perímetro␣es', perímetro)
elif opción == 'c':
    área = pi * radio ** 2
    print('El␣área␣es', área)
```

Es habitual que los programas con menú repitan una y otra vez las acciones de presentación del listado de opciones, lectura de selección y ejecución del cálculo. Una opción del menú permite finalizar el programa. Aquí tienes una nueva versión de `circulo.py` que finaliza cuando el usuario desea:

circulo.py

```
from math import pi

radio = float(input('Dame␣el␣radio␣de␣un␣círculo:␣'))

opción = ''
while opción != 'd':
    print('Escoge␣una␣opción:␣')
    print('a)␣Calcular␣el␣diámetro.')
    print('b)␣Calcular␣el␣perímetro.')
    print('c)␣Calcular␣el␣área.')
    print('d)␣Finalizar.')
    opción = input('Teclea␣a,␣b,␣c␣o␣d␣y␣pulsa␣el␣retorno␣de␣carro:␣')
    if opción == 'a':
        diámetro = 2 * radio
        print('El␣diámetro␣es', diámetro)
    elif opción == 'b':
        perímetro = 2 * pi * radio
        print('El␣perímetro␣es', perímetro)
    elif opción == 'c':
        área = pi * radio ** 2
        print('El␣área␣es', área)
    elif opción != 'd':
        print('Solo␣hay␣cuatro␣opciones:␣a,␣b,␣c␣o␣d.␣Tú␣has␣tecleado', opción)

print('Gracias␣por␣usar␣el␣programa')
```

▶ 114 El programa anterior pide el valor del radio al principio y, después, permite seleccionar uno o más cálculos con ese valor del radio. Modifica el programa para que pida el valor del radio cada vez que se solicita efectuar un nuevo cálculo.

▶ 115 Un vector en un espacio tridimensional es una tripleta de valores reales (x, y, z). Deseamos confeccionar un programa que permita operar con dos vectores. El usuario verá en pantalla un menú con las siguientes opciones:

```
1) Introducir el primer vector
2) Introducir el segundo vector
3) Calcular la suma
4) Calcular la diferencia
5) Calcular el producto escalar
6) Calcular el producto vectorial
7) Calcular el ángulo (en grados) entre ellos
8) Calcular la longitud
9) Finalizar
```

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

Puede que necesites que te refresquemos la memoria sobre los cálculos a realizar. Si es así, la tabla 4.1 te será de ayuda:

Operación	Cálculo
Suma: $(x_1, y_1, z_1) + (x_2, y_2, z_2)$	$(x_1 + x_2, y_1 + y_2, z_1 + z_2)$
Diferencia: $(x_1, y_1, z_1) - (x_2, y_2, z_2)$	$(x_1 - x_2, y_1 - y_2, z_1 - z_2)$
Producto escalar: $(x_1, y_1, z_1) \cdot (x_2, y_2, z_2)$	$x_1x_2 + y_1y_2 + z_1z_2$
Producto vectorial: $(x_1, y_1, z_1) \times (x_2, y_2, z_2)$	$(y_1z_2 - z_1y_2, z_1x_2 - x_1z_2, x_1y_2 - y_1x_2)$
Ángulo entre (x_1, y_1, z_1) y (x_2, y_2, z_2)	$\frac{180}{\pi} \cdot \arccos\left(\frac{x_1x_2 + y_1y_2 + z_1z_2}{\sqrt{x_1^2 + y_1^2 + z_1^2}\sqrt{x_2^2 + y_2^2 + z_2^2}}\right)$
Longitud de (x, y, z)	$\sqrt{x^2 + y^2 + z^2}$

Tabla 4.1: Recordatorio de operaciones básicas sobre vectores.

Tras la ejecución de cada una de las acciones del menú este reaparecerá en pantalla, a menos que la opción escogida sea la número 9. Si el usuario escoge una opción diferente, el programa advertirá al usuario de su error y el menú reaparecerá.

Las opciones 4 y 6 del menú pueden proporcionar resultados distintos en función del orden de los operandos, así que, si se escoge cualquiera de ellas, deberá mostrarse un nuevo menú que permita seleccionar el orden de los operandos. Por ejemplo, la opción 4 mostrará el siguiente menú:

```
1) Primer vector menos segundo vector
2) Segundo vector menos primer vector
```

Nuevamente, si el usuario se equivoca, se le advertirá del error y se le permitirá corregirlo.

La opción 8 del menú principal conducirá también a un submenú para que el usuario decida sobre cuál de los dos vectores se aplica el cálculo de longitud.

Ten en cuenta que tu programa debe contemplar y controlar toda posible situación excepcional: divisiones por cero, raíces con argumento negativo, etcétera. (Nota: La función arcocoseno se encuentra disponible en el módulo *math* y su identificador es *acos*).

4.2.5. El bucle for-in

Hay otro tipo de bucle en Python: el bucle **for-in**, que se puede leer como «para todo elemento de una serie, hacer...». Un bucle **for-in** presenta el siguiente aspecto:

```
for variable in serie de valores:
    acción
    acción
    ...
    acción
```

Veamos cómo funciona con un sencillo ejemplo:

```
saludos.py
for nombre in ['Pepe', 'Ana', 'Juan']:
    print('Hola,␣{0}.'.format(nombre))
```

Fíjate en que la relación de nombres va encerrada entre corchetes y que cada nombre se separa del siguiente con una coma. Se trata de una *lista* de nombres. Más adelante estudiaremos con detalle las listas. Ejecutemos ahora el programa. Por pantalla aparecerá el siguiente texto:

```
Hola, Pepe.
Hola, Ana.
Hola, Juan.
```

Se ha ejecutado la sentencia más sangrada una vez por cada valor de la serie de nombres y, con cada iteración, la variable *nombre* ha tomado el valor de uno de ellos (ordenadamente, de izquierda a derecha).

Estudia este programa:

```
potencias.py
número = int(input('Dame un número: '))

print('{0} elevado a {1} es {2}'.format(número, 2, número ** 2))
print('{0} elevado a {1} es {2}'.format(número, 3, número ** 3))
print('{0} elevado a {1} es {2}'.format(número, 4, número ** 4))
print('{0} elevado a {1} es {2}'.format(número, 5, número ** 5))
```

Podemos ofrecer una versión más simple:

```
potencias.py
número = int(input('Dame un número: '))

for potencia in [2, 3, 4, 5]:
    print('{0} elevado a {1} es {2}'.format(número, potencia, número ** potencia))
```

El bucle se lee de forma natural como «para toda *potencia* en la serie de valores 2, 3, 4 y 5, haz. . . ».

▶ 116 Haz un programa que muestre la tabla de multiplicar de un número introducido por teclado por el usuario. Aquí tienes un ejemplo de cómo se debe comportar el programa:

```
Dame un número: 5
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
5 x 10 = 50
```

▶ 117 Realiza un programa que proporcione el desglose en billetes y monedas de una cantidad entera de euros. Recuerda que hay billetes de 500, 200, 100, 50, 20, 10 y 5 € y monedas de 2 y 1 €. Debes «recorrer» los valores de billete y moneda disponibles con uno o más bucles **for-in**.

▶ 118 Haz un programa que muestre la raíz n-ésima de un número leído por teclado, para n tomando valores entre 2 y 100.

El último ejercicio propuesto es todo un desafío a nuestra paciencia: teclear 99 números separados por comas supone un esfuerzo bárbaro y conduce a un programa poco elegante.

Es hora de aprender una nueva función predefinida de Python que nos ayudará a evitar ese tipo de problemas: la función *range* (que en inglés significa «rango»). En principio, *range* se usa con dos argumentos: un *valor inicial* y un *valor final* (con matices). Si usamos *range* directamente veremos que proporciona un resultado curioso:

```
>>> range(0, 10)↵
range(0, 10)
```

La función *range* devuelve un objeto de tipo *range*, lo que resulta un tanto redundante. Un objeto de este tipo es un «enumerador» o «generador» y su sentido es muy dinámico: solo lo tiene cuando se usa para generar una secuencia de valores. Estudia este ejemplo:

```
contador_con_for.py
for i in range(1, 6):
    print(i)
```

Al ejecutar el programa, veremos lo siguiente por pantalla:

```
1
2
3
4
5
```

La secuencia de números que genera *range* es usada por el bucle **for-in** como serie de valores a recorrer. Observa que la serie de valores generados comprende todos los enteros entre los argumentos de la función, incluyendo al primero *pero no al último*.

Hay una función especial que toma como argumento una secuencia cualquiera de valores (y lo que devuelve *range* lo es) y construye con ella una lista: *list*. Viene bien para comprobar rápidamente lo que devuelve *range* si en algún momento tienes dudas:

```
>>> list(range(2, 10))↵
[2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(0, 3))↵
[0, 1, 2]
>>> list(range(-3, 3))↵
[-3, -2, -1, 0, 1, 2]
>>> list(range(-10, -1))↵
[-10, -9, -8, -7, -6, -5, -4, -3, -2]
```

El último ejercicio propuesto era pesadísimo: ¡nos obligaba a escribir una serie de 99 números! Con *range* resulta muchísimo más sencillo. He aquí la solución:

```
raices.py
número = float(input('Dame␣un␣número:␣'))

for n in range(2, 101):
    print('La␣raíz␣{0}-ésima␣de␣{1}␣es␣{2}'.format(n, número, número**(1/n)))
```

Fíjate en que *range* tiene por segundo argumento el valor 101 y no 100: recuerda que con *range* el último elemento de la lista no llega a ser el valor final.

Podemos utilizar la función *range* con uno, dos o tres argumentos. Si usamos *range* con un argumento estaremos especificando únicamente el último valor (más uno) de la serie, pues el primero vale 0 por defecto:

```
>>> list(range(5))↵
[0, 1, 2, 3, 4]
```

Si usamos tres argumentos, el tercero permite especificar un *incremento* para la serie de valores. Observa en estos ejemplos qué listas de enteros devuelve *range*:

```
>>> list(range(2, 10, 2))↵
[2, 4, 6, 8]
>>> list(range(2, 10, 3))↵
[2, 5, 8]
```

Fíjate en que si pones un incremento negativo (un *decremento*), la lista va de los valores altos a los bajos:

```
>>> list(range(10, 5, -1))↵
[10, 9, 8, 7, 6]
>>> list(range(3, -1, -1))↵
[3, 2, 1, 0]
>>> list(range(10, 1, -3))↵
[10, 7, 4]
```

Así pues, si el tercer argumento es negativo, la lista finaliza con un valor *mayor que* el segundo argumento (y no menor).

Finalmente, observa que es equivalente utilizar *range* con dos argumentos a utilizarla con un valor del incremento igual a 1.

```
>>> list(range(2, 5, 1))↵
[2, 3, 4]
>>> list(range(2, 5))↵
[2, 3, 4]
```

▶ 119 Haz un programa que muestre, en líneas independientes, todos los números pares comprendidos entre 0 y 200 (ambos inclusive).

▶ 120 Haz un programa que muestre, en líneas independientes y en orden inverso, todos los números pares comprendidos entre 0 y 200 (ambos inclusive).

▶ 121 Escribe un programa que muestre los números pares positivos entre 2 y un número cualquiera que introduzca el usuario por teclado.

Obi Wan

Puede resultar sorprendente que *range*(*a*, *b*) incluya todos los números enteros comprendidos entre *a* y *b*, pero sin incluir *b*. En realidad la forma «natural» o más frecuente de usar *range* es con un solo parámetro, *range*(*n*), que devuelve una lista con los *n* primeros números enteros incluyendo al cero (hay razones para que esto sea lo conveniente, ya llegaremos). Como incluye al cero y hay *n* números, no puede incluir al propio número *n*. Al extenderse el uso de *range* a dos argumentos, se ha mantenido la «compatibilidad» eliminando el último elemento. Una primera ventaja es que resulta fácil calcular cuántas iteraciones realizará un bucle *range*(*a*, *b*): exactamente $b - a$. (Si el valor *b* estuviera incluido, el número de elementos sería $b - a + 1$).

Hay que ir con cuidado, pues es fácil equivocarse «por uno». De hecho, equivocarse «por uno» es tan frecuente al programar (y no solo con *range*) que hay una expresión para este tipo de error: un error Obi Wan (Kenobi), que es más o menos como suena en inglés «off by one» (pasarse o quedarse corto por uno).

4.2.6. for-in como forma compacta de ciertos while

Ciertos bucles se ejecutan un número de veces fijo y conocido *a priori*. Por ejemplo, al desarrollar el programa que calcula el sumatorio de los 1000 primeros números utilizamos un bucle que iteraba exactamente 1000 veces:

sumatorio.py

```
sumatorio = 0
i = 1
while i <= 1000:
    sumatorio += i
    i += 1
print(sumatorio)
```

El bucle se ha construido de acuerdo con un patrón, una especie de «frase hecha» del lenguaje de programación:

```
i = valor inicial
while i <= valor final:
    acciones
    i += 1
```

En este patrón la variable *i* suele denominarse *índice* del bucle.

Podemos expresar de forma compacta este tipo de bucles con un **for-in** siguiendo este otro patrón:

```
for i in range(valor inicial, valor final + 1):
    acciones
```

Fíjate en que las cuatro líneas del fragmento con **while** pasan a expresarse con solo dos gracias al **for-in** con *range*.

El programa de cálculo del sumatorio de los 1000 primeros números se puede expresar ahora de este modo:

sumatorio.py
```
sumatorio = 0
for i in range(1, 1001):
    sumatorio += i

print(sumatorio)
```

¡Bastante más fácil de leer que usando un **while**!

▶ 122 Haz un programa que pida el valor de dos enteros n y m y que muestre por pantalla el valor de

$$\sum_{i=n}^{m} i.$$

Debes usar un bucle **for-in** para el cálculo del sumatorio.

▶ 123 Haz un programa que pida el valor de dos enteros n y m y que muestre por pantalla el valor de

$$\sum_{i=n}^{m} i^2.$$

▶ 124 Haz un programa que pida el valor de dos enteros n y m y calcule el sumatorio de todos los números pares comprendidos entre ellos (incluyéndolos en el caso de que sean pares).

4.2.7. Números primos

Vamos ahora con un ejemplo más. Nos proponemos construir un programa que nos diga si un número (entero) es o no es *primo*. Recuerda: un número primo es aquel número mayor que 1 que solo es divisible por 1 y por sí mismo.

¿Cómo empezar? Resolvamos un problema concreto, a ver qué estrategia seguiríamos normalmente. Supongamos que deseamos saber si 7 es primo. Podemos intentar dividirlo por cada uno de los números entre 2 y 6. Si alguna de las divisiones es exacta, entonces el número *no* es primo:

Dividendo	Divisor	Cociente	Resto
7	2	3	1
7	3	2	1
7	4	1	3
7	5	1	2
7	6	1	1

Ahora estamos seguros: ninguno de los restos dio 0, así que 7 es primo. Hagamos que el ordenador nos muestre esa misma tabla:

es_primo.py
```
número = 7

for divisor in range(2, número):
```

```
    print('{0}␣entre␣{1}␣'.format(número, divisor), end='')
    print('es␣{0}␣con␣resto␣{1}'.format(número // divisor, número % divisor))
```

(Recuerda que *range*(2, *número*) genera todos los números enteros comprendidos entre 2 y *número* - 1). Aquí tienes el resultado de ejecutar el programa:

```
7 entre 2 es 3 con resto 1
7 entre 3 es 2 con resto 1
7 entre 4 es 1 con resto 3
7 entre 5 es 1 con resto 2
7 entre 6 es 1 con resto 1
```

Está claro que probar todas las divisiones es fácil, pero, ¿cómo nos aseguramos de que *todos* los restos son distintos de cero? Una posibilidad es contarlos y comprobar que hay exactamente *número* - 2 restos no nulos:

```
es_primo.py
número = 7

restos_no_nulos = 0
for divisor in range(2, número):
    if número % divisor != 0:
        restos_no_nulos += 1

if restos_no_nulos == número - 2:
    print('El␣número␣{0}␣es␣primo.'.format(número))
else:
    print('El␣número␣{0}␣no␣es␣primo.'.format(número))
```

Pero vamos a proponer un método distinto basado en una «idea feliz» y que, más adelante, nos permitirá acelerar notabilísimamente el cálculo. Vale la pena que la estudies bien: la utilizarás siempre que quieras probar que *toda una serie* de valores cumple una propiedad. En nuestro caso, la propiedad que queremos demostrar que cumplen todos los números comprendidos entre 2 y *número* - 1 es «al dividir a *número*, da resto distinto de cero».

- Empieza siendo optimista: supón que la propiedad es cierta y asigna a una variable el valor «cierto».
- Recorre todos los números y cuando alguno de los elementos de la secuencia no satisfaga la propiedad, modifica la variable antes mencionada para que contenga el valor «falso».
- Al final de todo, mira qué vale la variable: si aún vale «cierto», es que nadie la puso a «falso», así que la propiedad se cumple para todos los elementos y el número es primo; y si vale «falso», entonces alguien la puso a «falso» y para eso es preciso que algún elemento no cumpliera la propiedad en cuestión, por lo que el número no puede ser primo.

Mira cómo plasmamos esa idea en un programa:

```
es_primo.py
número = 7

creo_que_es_primo = True
for divisor in range(2, número):
    if número % divisor == 0:
        creo_que_es_primo = False

if creo_que_es_primo:
    print('El␣número␣{0}␣es␣primo.'.format(número))
else:
    print('El␣número␣{0}␣no␣es␣primo.'.format(número))
```

▶ 125 Haz un traza del programa para los siguientes valores de la variable *número*:

a) 4

b) 13

c) 25

d) 2

True == True

Fíjate en la línea 8 de este programa:

```
es_primo.py
1  número = 7
2
3  creo_que_es_primo = True
4  for divisor in range(2, número):
5      if número % divisor == 0:
6          creo_que_es_primo = False
7
8  if creo_que_es_primo:
9      print('El número {0} es primo.'.format(número))
10 else:
11     print('El número {0} no es primo.'.format(número))
```

La condición del if es muy extraña, ¿no? No hay comparación alguna. ¿Qué condición es esa? Muchos estudiantes optan por esta fórmula alternativa para las líneas 8 y similares:

```
es_primo.py
1  número = 7
2
3  creo_que_es_primo = True
4  for divisor in range(2, número):
5      if número % divisor == 0:
6          creo_que_es_primo = False
7
8  if creo_que_es_primo == True:
9      print('El número {0} es primo.'.format(número))
10 else:
11     print('El número {0} no es primo.'.format(número))
```

Les parece más natural porque de ese modo se compara el valor de *creo_que_es_primo* con algo. Pero, si lo piensas bien, esa comparación es superflua: a fin de cuentas, el resultado de la comparación *creo_que_es_primo* == True es True, precisamente lo que ya vale *creo_que_es_primo*.

No es que esté mal efectuar esa comparación extra, sino que no aporta nada y resta legibilidad. Evítala si puedes.

Después de todo, no es tan difícil. Aunque esta idea feliz la utilizarás muchas veces, es probable que cometas un error (al menos, muchos compañeros tuyos caen en él una y otra vez). Fíjate en este programa, que está mal:

```
es_primo.py
número = 7

creo_que_es_primo = True
for divisor in range(2, número):
    if número % divisor == 0:
        creo_que_es_primo = False
    else:
        creo_que_es_primo = True

if creo_que_es_primo:
```

```
11     print('El número {0} es primo.'.format(número))
12 else:
13     print('El número {0} no es primo.'.format(número))
```

¡El programa solo se acuerda de lo que pasó con el último valor del bucle! Haz la prueba: haz una traza sustituyendo la asignación de la línea 1 por la sentencia *número* = 4. El número *no* es primo, pero al no ser exacta la división entre 4 y 3 (el último valor de *divisor* en el bucle), el valor de *creo_que_es_primo* es True. El programa concluye, pues, que 4 es primo.

Se cumple para todos / se cumple para alguno

Muchos de los programas que diseñaremos necesitan verificar que cierta condición se cumple para algún elemento de un conjunto o para todos los elementos del conjunto. En ambos casos tendremos que recorrer todos los elementos, uno a uno, y comprobar si la condición es cierta o falsa para cada uno de ellos.

Cuando queramos comprobar que *todos* cumplen una condición, haremos lo siguiente:

1) Seremos *optimistas* y empezaremos suponiendo que la condición se cumple para todos.
2) Preguntaremos a cada uno de los elementos si cumple la condición.
3) Solo cuando detectemos que uno de ellos *no la cumple*, cambiaremos de opinión y pasaremos a saber que la condición no se cumple para todos. *Nada nos podrá hacer cambiar de opinión.*

He aquí un esquema que usa la notación de Python:

```
creo_que_se_cumple_para_todos = True
for elemento in conjunto:
    if not condición:
        creo_que_se_cumple_para_todos = False
if creo_que_se_cumple_para_todos:
    print('Se cumple para todos')
```

Cuando queramos comprobar que *alguno* cumple una condición, haremos lo siguiente:

1) Seremos *pesimistas* y empezaremos suponiendo que la condición no se cumple para ninguno.
2) Preguntaremos a cada uno de los elementos si se cumple la condición.
3) Solo cuando detectemos que uno de ellos *sí la cumple*, cambiaremos de opinión y pasaremos a saber que la condición se cumple para alguno. *Nada nos podrá hacer cambiar de opinión.*

He aquí un esquema que usa la notación de Python:

```
creo_que_se_cumple_para_alguno = False
for elemento in conjunto:
    if condición:
        creo_que_se_cumple_para_alguno = True
if creo_que_se_cumple_para_alguno:
    print('Se cumple para alguno')
```

Vamos a refinar el programa. En primer lugar, haremos que trabaje con cualquier número que el usuario introduzca:

es_primo.py
```
número = int(input('Dame un número: '))

creo_que_es_primo = True
for divisor in range(2, número):
    if número % divisor == 0:
        creo_que_es_primo = False

if creo_que_es_primo:
    print('El número {0} es primo.'.format(número))
else:
```

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1   Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

```
    print('El número {0} no es primo.'.format(número))
```

El programa presenta un punto débil: cuando *número* toma el valor 1, el resultado proporcionado es incorrecto:

```
Dame un número: 1↵
El número 1 es primo.
```

El número no es primo, pero al no ejecutarse ninguna vez el bucle **for-in**, el valor de *creo_que_es_primo* sigue siendo True. La solución es fácil:

```
es_primo.py
número = int(input('Dame un número: '))

if número > 1:
    creo_que_es_primo = True
    for divisor in range(2, número):
        if número % divisor == 0:
            creo_que_es_primo = False
else:
    creo_que_es_primo = False

if creo_que_es_primo:
    print('El número {0} es primo.'.format(número))
else:
    print('El número {0} no es primo.'.format(número))
```

▶ 126 ¿Sería correcta la siguiente versión del programa?

```
es_primo.py
número = int(input('Dame un número: '))

if número > 1:
    creo_que_es_primo = True
else:
    creo_que_es_primo = False
for divisor in range(2, número):
    if número % divisor == 0:
        creo_que_es_primo = False

if creo_que_es_primo:
    print('El número {0} es primo.'.format(número))
else:
    print('El número {0} no es primo.'.format(número))
```

Ahora vamos a hacer que el programa vaya más rápido. Observa qué ocurre cuando tratamos de ver si el número 1024 es primo o no. Empezamos dividiéndolo por 2 y vemos que el resto de la división es cero. Pues ya está: estamos seguros de que 1024 no es primo. Sin embargo, nuestro programa sigue haciendo cálculos: pasa a probar con el 3, y luego con el 4, y con el 5, y así hasta llegar al 1023. ¿Para qué, si ya sabemos que no es primo? Nuestro objetivo es que el bucle deje de ejecutarse tan pronto estemos seguros de que el número no es primo. Pero resulta que no podemos hacerlo con un bucle **for-in**, pues este tipo de bucles se basa en nuestro conocimiento *a priori* de cuántas iteraciones vamos a hacer. Como en este caso no lo sabemos, hemos de utilizar un bucle **while**. Escribamos primero un programa equivalente al anterior, pero usando un **while** en lugar de un **for-in**:

```
es_primo.py
número = int(input('Dame un número: '))

```

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

```
if número > 1:
    creo_que_es_primo = True
    divisor = 2
    while divisor < número:
        if número % divisor == 0:
            creo_que_es_primo = False
        divisor += 1
else:
    creo_que_es_primo = False

if creo_que_es_primo:
    print('El número {0} es primo.'.format(número))
else:
    print('El número {0} no es primo.'.format(número))
```

▶ 127 Haz una traza del último programa para el número 125.

Error para alguno / error para todos

Ya te hemos dicho que muchos de los programas que diseñaremos necesitan verificar que cierta condición se cumple para algún elemento de un conjunto o para todos los elementos del conjunto. Y también te hemos dicho cómo abordar ambos problemas. Pero, aun así, es probable que cometas un error (muchos, muchos estudiantes lo hacen). Aquí tienes un ejemplo de programa erróneo al tratar de comprobar que una condición se cumple para todos los elementos de un conjunto:

```
creo_que_se_cumple_para_todos = True
for elemento in conjunto:
    if not condición:
        creo_que_se_cumple_para_todos = False
    else: # Esta línea y la siguiente sobran
        creo_que_se_cumple_para_todos = True

if creo_que_se_cumple_para_todos:
    print('Se cumple para todos')
```

Y aquí tienes una versión errónea para el intento de comprobar que una condición se cumple para alguno:

```
creo_que_se_cumple_para_alguno = False
for elemento in conjunto:
    if condición:
        creo_que_se_cumple_para_alguno = True
    else: # Esta línea y la siguiente sobran
        creo_que_se_cumple_para_alguno = False

if creo_que_se_cumple_para_alguno:
    print('Se cumple para alguno')
```

En ambos casos, solo se está comprobando si *el último* elemento del conjunto cumple o no la condición.

Hemos sustituido el **for-in** por un **while**, pero no hemos resuelto el problema: con el 1024 seguimos haciendo todas las pruebas de divisibilidad. ¿Cómo hacer que el bucle acabe tan pronto se esté seguro de que el número no es primo? Pues complicando un poco la condición del **while**:

```
es_primo.py
número = int(input('Dame un número: '))

if número > 1:
    creo_que_es_primo = True
```

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

```
    divisor = 2
    while divisor < número and creo_que_es_primo:
        if número % divisor == 0:
            creo_que_es_primo = False
        divisor += 1
else:
    creo_que_es_primo = False

if creo_que_es_primo:
    print('El␣número␣{0}␣es␣primo.'.format(número))
else:
    print('El␣número␣{0}␣no␣es␣primo.'.format(número))
```

Ahora sí.

▶ 128 Haz una traza del último programa para el número 125.

▶ 129 Haz un programa que calcule el máximo común divisor (mcd) de dos enteros positivos. El mcd es el número más grande que divide exactamente a ambos números.

▶ 130 Haz un programa que calcule el máximo común divisor (mcd) de tres enteros positivos. El mcd de tres números es el número más grande que divide exactamente a los tres.

4.2.8. Rotura de bucles: break

El último programa diseñado aborta su ejecución tan pronto sabemos que el número estudiado no es primo. La variable *creo_que_es_primo* juega un doble papel: «recordar» si el número es primo o no al final del programa y abortar el bucle **while** tan pronto sabemos que el número no es primo. La condición del **while** se ha complicado un poco para tener en cuenta el valor de *creo_que_es_primo* y abortar el bucle inmediatamente.

Hay una sentencia que permite abortar la ejecución de un bucle desde cualquier punto del mismo: **break** (en inglés significa «romper»). Observa esta nueva versión del mismo programa:)

es_primo.py
```
1 número = int(input('Dame␣un␣número:␣'))
2 
3 if número > 1:
4     creo_que_es_primo = True
5     divisor = 2
6     while divisor < número:
7         if número % divisor == 0:
8             creo_que_es_primo = False
9             break
10        divisor += 1
11 else:
12     creo_que_es_primo = False
13 
14 if creo_que_es_primo:
15     print('El␣número␣{0}␣es␣primo.'.format(número))
16 else:
17     print('El␣número␣{0}␣no␣es␣primo.'.format(número))
```

Cuando se ejecuta la línea 9, el programa sale inmediatamente del bucle, es decir, pasa a la línea 13 sin pasar por la línea 10.

Nuevamente estamos ante una comodidad ofrecida por el lenguaje: la sentencia **break** permite expresar de otra forma una idea que ya podía expresarse sin ella. Solo debes considerar la utilización de **break** cuando te resulte cómoda. No abuses del **break**: a veces, una condición bien expresada en la primera línea del bucle **while** hace más legible un programa.

La sentencia **break** también es utilizable con el bucle **for-in**. Analicemos esta nueva versión de `es_primo.py`:

```
es_primo.py
número = int(input('Dame un número: '))

if número > 1:
    creo_que_es_primo = True
    for divisor in range(2, número):
        if número % divisor == 0:
            creo_que_es_primo = False
            break
else:
    creo_que_es_primo = False

if creo_que_es_primo:
    print('El número {0} es primo.'.format(número))
else:
    print('El número {0} no es primo.'.format(número))
```

Esta versión es más concisa que la anterior (ocupa menos líneas) y, en cierto sentido, más elegante: el bucle **for-in** expresa mejor la idea de que *divisor* recorre ascendentemente un rango de valores.

Versiones eficientes de «se cumple para alguno / se cumple para todos»

Volvemos a visitar los problemas de «se cumple para alguno» y «se cumple para todos». Esta vez vamos a hablar de cómo acelerar el cálculo gracias a la sentencia **break**.

Si quieres comprobar si una condición se cumple para todos los elementos de un conjunto y encuentras que uno de ellos no la satisface, ¿para qué seguir? ¡Ya sabemos que no se cumple para todos!

```
creo_que_se_cumple_para_todos = True
for elemento in conjunto:
    if not condición:
        creo_que_se_cumple_para_todos = False
        break

if creo_que_se_cumple_para_todos:
    print('Se cumple para todos')
```

Como ves, esta mejora puede suponer una notable aceleración del cálculo: cuando el primer elemento del conjunto no cumple la condición, acabamos inmediatamente. Ese es el mejor de los casos. El peor de los casos es que todos cumplan la condición, pues nos vemos obligados a recorrer todos los elementos del conjunto. Y eso es lo que hacíamos hasta el momento: recorrer todos los elementos. O sea, en el peor de los casos, hacemos el mismo esfuerzo que veníamos haciendo para todos los casos. ¡No está nada mal!

Si quieres comprobar si una condición se cumple para alguno de los elementos de un conjunto y encuentras que uno de ellos la satisface, ¿para qué seguir? ¡Ya sabemos que la cumple alguno!

```
creo_que_se_cumple_para_alguno = False
for elemento in conjunto:
    if condición:
        creo_que_se_cumple_para_alguno = True
        break

if creo_que_se_cumple_para_alguno:
    print('Se cumple para alguno')
```

Podemos hacer la misma reflexión en torno a la eficiencia de esta nueva versión que en el caso anterior.

▶ 131 Haz una traza del programa para el valor 125.

▶ 132 En realidad no hace falta explorar todo el rango de números entre 2 y $n-1$ para saber si un número n es o no es primo. Basta con explorar el rango de números entre 2 y la parte entera de $n/2$. Piensa por qué. Modifica el programa para que solo exploremos ese rango.

▶ 133 Ni siquiera hace falta explorar todo el rango de números entre 2 y $n/2$ para saber si un número n es o no es primo. Basta con explorar el rango de números entre 2 y la parte entera de $\sqrt{n}$. (Créetelo). Modifica el programa para que solo exploremos ese rango.

4.2.9. Anidamiento de estructuras

Ahora vamos a resolver otro problema. Vamos a hacer que el programa pida un número y nos muestre por pantalla los números primos entre 1 y el que hemos introducido. Mira este programa:

primo.py
```
1  límite = int(input('Dame un número: '))
2
3  for número in range(2, límite+1):
4      creo_que_es_primo = True
5      for divisor in range(2, número):
6          if número % divisor == 0:
7              creo_que_es_primo = False
8              break
9      if creo_que_es_primo:
10         print(número, end=' ')
11 print()
```

No debería resultarte difícil entender el programa. Tiene *bucles anidados* (un **for-in** dentro de un **for-in**), pero está claro qué hace cada uno de ellos: el más exterior recorre con *número* todos los números comprendidos entre 2 (pues ya sabemos que 1 no es primo) y *límite*, ambos inclusive; el más interior forma parte del procedimiento que determina si el número que estamos estudiando en cada instante es o no es primo.

Dicho de otro modo: *número* va tomando valores entre 2 y *límite* y *para cada valor* de *número* se ejecuta el bloque de las líneas 4–10, así que, *para cada* valor de *número*, se comprueba si este es primo o no. Solo si el número resulta ser primo se muestra por pantalla.

Puede que te intrigue el **break** de la línea 8. ¿A qué bucle «rompe»? Solo al más interior: una sentencia **break** siempre aborta la ejecución de un solo bucle y este es el que la contiene directamente.

Probemos el programa

```
Dame un número: 100↵
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

Antes de acabar: existen procedimientos más eficientes para determinar si un número es primo o no, así como para listar los números primos en un intervalo. Hacer buenos programas no solo pasa por conocer bien las reglas de escritura de programas en un lenguaje de programación: has de saber diseñar algoritmos y, muchas veces, buscar los mejores algoritmos conocidos en los libros.

▶ 134 ¿Qué resultará de ejecutar estos programas?

1) **ejercicio_for.py**
```
for i in range(0, 5):
    for j in range(0, 3):
        print(i, j)
```

Índice de bucle for-in: ¡prohibido asignar!

Hemos aprendido que el bucle **for-in** utiliza una variable índice a la que se van asignando los diferentes valores del rango. En muchos ejemplos se utiliza la variable *i*, pero solo porque también en matemáticas los sumatorios y productorios suelen utilizar la letra *i* para indicar el nombre de su variable índice. Puedes usar cualquier nombre de variable válido.

Pero que el índice sea una variable cualquiera no te da libertad absoluta para hacer con ella lo que quieras. En un bucle **for-in**, las variables de índice solo deben usarse para consultar su valor, nunca para asignarles uno nuevo. Por ejemplo, este fragmento de programa es incorrecto:

```
for i in range(0, 5):
    i += 2
```

Y ahora que sabes que los bucles pueden anidarse, también has de tener mucho cuidado con sus índices. Un error frecuente entre primerizos de la programación es utilizar el mismo índice para dos bucles anidados. Por ejemplo, estos bucles anidados están mal:

```
for i in range(0, 5):
    for i in range(0, 3):
        print(i)
```

En el fondo, este problema es una variante del anterior, pues de algún modo se está asignando nuevos valores a la variable *i* en el bucle interior, pero *i* es la variable del bucle exterior y asignarle cualquier valor está prohibido.

Recuerda: *nunca debes asignar un valor a un índice de bucle* **for-in** *ni usar la misma variable índice en bucles anidados.*

2) ejercicio_for.py
```
for i in range(0, 5):
    for j in range(i, 5):
        print(i, j)
```

3) ejercicio_for.py
```
for i in range(0, 5):
    for j in range(0, i):
        print(i, j)
```

4) ejercicio_for.py
```
for i in range(0, 4):
    for j in range(0, 4):
        for k in range(0, 2):
            print(i, j, k)
```

5) ejercicio_for.py
```
for i in range(0, 4):
    for j in range(0, 4):
        for k in range(i, j):
            print(i, j, k)
```

6) ejercicio_for.py
```
for i in range(1, 5):
    for j in range(0, 10, i):
        print(i, j)
```

Una excepción a la regla de sangrado

Cada vez que una sentencia acaba con dos puntos (:), Python espera que la sentencia o sentencias que le siguen aparezcan con un mayor sangrado. Es la forma de marcar el inicio y el fin de una serie de sentencias que «dependen» de otra.

Hay una excepción: si solo hay *una* sentencia que «depende» de otra, puedes escribir ambas en la misma línea. Este programa:

```
a = int(input('Dame un entero positivo: '))
while a < 0:
    a = int(input('Te he dicho positivo: '))
if a % 2 == 0:
    print('El número es par')
else:
    print('El número es impar')
```

y este otro:

```
a = int(input('Dame un entero positivo: '))
while a < 0: a = int(input('Te he dicho positivo: '))
if a % 2 == 0: print('El número es par')
else: print('El número es impar')
```

son equivalentes, aunque el primero resulta más legible.

4.3. Captura y tratamiento de excepciones

Ya has visto que en nuestros programas pueden aparecer errores en tiempo de ejecución, es decir, pueden generar *excepciones*: divisiones por cero, intentos de calcular raíces de valores negativos, problemas al operar con tipos incompatibles (como al sumar una cadena y un entero), etc. Hemos presentado la estructura de control if como un medio para controlar estos problemas y ofrecer un tratamiento especial cuando convenga (aunque luego hemos considerado muchas otras aplicaciones de esta sentencia). En ocasiones, la detección de posibles errores con if resulta un tanto pesada, pues modifica sensiblemente el aspecto del programa al llenarlo de comprobaciones.

Hay una estructura de control especial para la detección y tratamiento de excepciones: try-except. Su forma básica de uso es esta:

```
try:
    acción potencialmente errónea
    acción potencialmente errónea
    ...
    acción potencialmente errónea
except:
    acción para tratar el error
    acción para tratar el error
    ...
    acción para tratar el error
```

Podemos expresar la idea fundamental así:

«Intenta ejecutar estas acciones y, si se comete un error, ejecuta inmediatamente estas otras».

Es fácil entender qué hace básicamente si estudiamos un ejemplo sencillo. Volvamos a considerar el problema de la resolución de una ecuación de primer grado:

primer_grado.py
```
a = float(input('Valor de a: '))
b = float(input('Valor de b: '))

```

```
4 try:
5     x = -b/a
6     print('Solución: ', x)
7 except:
8     if b != 0:
9         print('La ecuación no tiene solución.')
10    else:
11        print('La ecuación tiene infinitas soluciones.')
```

Las líneas 5 y 6 están en un bloque que depende de la sentencia **try**. Es admisible que se lancen excepciones desde ese bloque. Si se lanza una, la ejecución pasará inmediatamente al bloque que depende de la sentencia **except**. Hagamos dos trazas, una para una configuración de valores de a y b que provoque un error de división por cero y una para otra que no genere excepción alguna:

$a = 0$ y $b = 3$	$a = 1$ y $b = -1$
Las líneas 1 y 2 se ejecutan, con lo que se leen los valores de a y b.	Las líneas 1 y 2 se ejecutan, con lo que se leen los valores de a y b.
La línea 4 se ejecuta, pero no hay un efecto asociado a su ejecución.	La línea 4 se ejecuta, pero no hay un efecto asociado a su ejecución.
Al ejecutarse la línea 5, se produce una excepción (división por cero). *Se salta inmediatamente a la línea 8.*	Se ejecutan las líneas 5 y 6, con lo que se muestra por pantalla el valor de la solución de la ecuación: `Solución: 1`. La ejecución finaliza.
Se ejecuta la línea 8 y el resultado de la comparación es *cierto*.	
La línea 9 se ejecuta y se muestra por pantalla el mensaje `La ecuación no tiene solución.`	

Atrevámonos ahora con la resolución de una ecuación de segundo grado:

segundo_grado.py
```
from math import sqrt

a = float(input('Valor de a: '))
b = float(input('Valor de b: '))
c = float(input('Valor de c: '))

try:
    x1 = (-b + sqrt(b**2 - 4*a*c)) / (2 * a)
    x2 = (-b - sqrt(b**2 - 4*a*c)) / (2 * a)
    if x1 == x2:
        print('Solución: x={0:.3f}'.format(x1))
    else:
        print('Soluciones: x1={0:.3f} y x2={1:.3f}'.format(x1, x2))
except:
    # No sabemos si llegamos aquí por una división por cero o si llegamos
    # por intentar calcular la raíz cuadrada de un discriminante negativo.
    print('O no hay soluciones reales o es una ecuación de primer grado')
```

Como es posible que se cometan dos tipos de error diferentes, al llegar al bloque dependiente del **except** no sabemos cuál de los dos tuvo lugar. Evidentemente, podemos efectuar las comprobaciones pertinentes sobre los valores de a, b y c para deducir el error concreto, pero queremos contarte otra posibilidad de la sentencia **try-except**. Las excepciones tienen un «tipo» asociado y podemos distinguir el tipo de excepción para actuar de diferente forma en función

del tipo de error detectado. Una división por cero es un error de tipo *ZeroDivisionError* y el intento de calcular la raíz cuadrada de un valor negativo es un error de tipo *ValueError*. Mmmm. Resulta difícil recordar de qué tipo es cada error, pero el intérprete de Python resulta útil para recordar si provocamos deliberadamente un error del tipo que deseamos tratar:

```
>>> 1 / 0
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ZeroDivisionError: division by zero
>>> from math import sqrt
>>> sqrt(-1)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: math domain error
```

Es posible usar varias cláusulas **except**, una por cada tipo de error a tratar:

segundo_grado.py
```
from math import sqrt

a = float(input('Valor de a: '))
b = float(input('Valor de b: '))
c = float(input('Valor de c: '))

try:
    x1 = (-b + sqrt(b**2 - 4*a*c)) / (2 * a)
    x2 = (-b - sqrt(b**2 - 4*a*c)) / (2 * a)
    if x1 == x2:
        print('Solución: x={0:.3f}'.format(x1))
    else:
        print('Soluciones: x1={0:.3f} y x2={1:.3f}'.format(x1, x2))
except ZeroDivisionError:
    if b != 0:
        print('La ecuación no tiene solución.')
    else:
        print('La ecuación tiene infinitas soluciones.')
except ValueError:
    print('No hay soluciones reales')
```

4.4. Algunos ejemplos gráficos

4.4.1. Un graficador de funciones.

Vamos a usar el módulo de la tortuga para representar gráficamente funciones matemáticas. Utilizaremos la función seno, pero trataremos de que nuestro programa sea fácilmente modificable para utilizar las que deseemos.

Como la función seno toma valores entre -1 y 1 y la vamos a representar en el intervalo entre -2π y 2π, vamos a empezar por definir la superficie de dibujo y el sistema de coordenadas para la tortuga.

seno.py
```
from turtle import Screen, Turtle
from math import sin, pi

pantalla = Screen()
pantalla.setup(825, 425)
pantalla.screensize(800, 400)
pantalla.setworldcoordinates(-2*pi, -1, 2*pi, 1)
```

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

Dibujemos algunos puntos de la función. Usaremos el método *goto* en lugar de giros con *left* o *right* y desplazamientos con *forward*, pues queremos dibujar la función a partir de coordenadas absolutas.

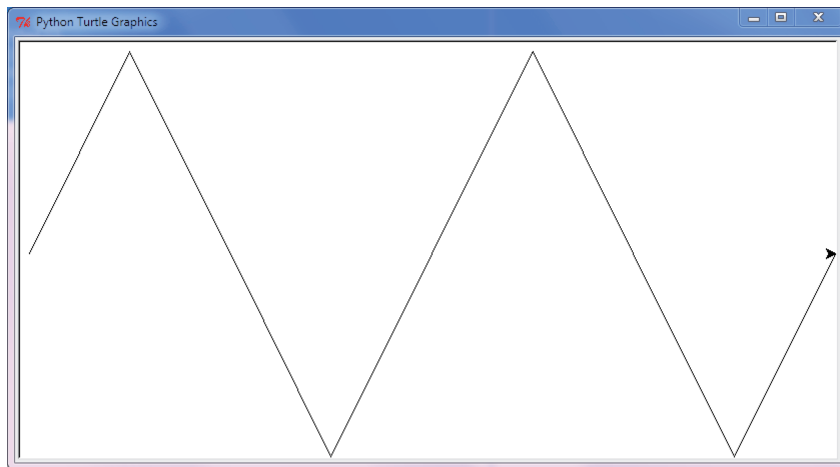
seno.py
```
from turtle import Screen, Turtle
from math import sin, pi

pantalla = Screen()
pantalla.setup(825, 425)
pantalla.screensize(800, 400)
pantalla.setworldcoordinates(-2*pi, -1, 2*pi, 1)

tortuga = Turtle()
tortuga.penup()
tortuga.goto(-2*pi,sin(-2*pi))
tortuga.pendown()
tortuga.goto(-1.5*pi,sin(-1.5*pi))
tortuga.goto(-1*pi,sin(-1*pi))
tortuga.goto(-0.5*pi,sin(-0.5*pi))
tortuga.goto(0,sin(0))
tortuga.goto(0.5*pi,sin(0.5*pi))
tortuga.goto(1*pi,sin(1*pi))
tortuga.goto(1.5*pi,sin(1.5*pi))
tortuga.goto(2*pi,sin(2*pi))

pantalla.exitonclick()
```

El resultado no es muy suave:



Aparecen pocos puntos, pero podemos apreciar que están dispuestos como corresponde a la función seno. La cosa mejoraría añadiendo más puntos, pero desde luego que no lo haremos repitiendo líneas en el programa como en el ejemplo: usaremos un bucle **while**.

La idea es hacer que una variable, digamos x, vaya recorriendo, paso a paso, el intervalo $[-2\pi, 2\pi]$, y para cada valor, llamar a `tortuga.goto(x, sin(x))`. ¿Qué queremos decir con «paso a paso»? Pues que de una iteración a la siguiente, aumentaremos x en una cantidad fija. Pongamos, inicialmente, que esta cantidad es 0.5. Nuestro programa presentará este aspecto:

seno.py
```
from turtle import Screen, Turtle
from math import sin, pi

pantalla = Screen()
pantalla.setup(825, 425)
```

```
pantalla.screensize(800, 400)
pantalla.setworldcoordinates(-2*pi, -1, 2*pi, 1)

tortuga = Turtle()

x = valor inicial
tortuga.penup()
tortuga.goto(x, sin(x))
tortuga.pendown()
while condición:
    tortuga.goto(x, sin(x))
    x += 0.5

pantalla.exitonclick()
```

¿Qué valor inicial asignamos a x? Podemos probar con -2π, que es la coordenada X del primer punto que nos interesa mostrar. ¿Y qué condición ponemos en el **while**? A ver, nos interesa repetir mientras x sea menor o igual que 2π. Pues ya está:

```
seno.py
from turtle import Screen, Turtle
from math import sin, pi

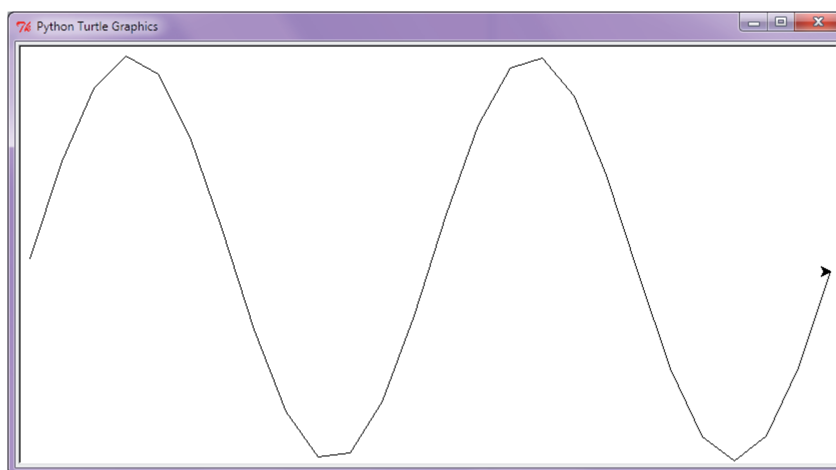
pantalla = Screen()
pantalla.setup(825, 425)
pantalla.screensize(800, 400)
pantalla.setworldcoordinates(-2*pi, -1, 2*pi, 1)

tortuga = Turtle()

x = -2*pi
tortuga.penup()
tortuga.goto(x, sin(x))
tortuga.pendown()
while x <= 2*pi:
    tortuga.goto(x, sin(x))
    x += 0.5

pantalla.exitonclick()
```

El resultado es ahora mucho más suave:



Aun así, nos gustaría mostrar más puntos. Ahora el cambio que debemos efectuar es muy sencillo: en lugar de poner un incremento de 0.5, podemos poner un incremento más pequeño. Cuanto

menor sea el incremento, más puntos dibujaremos. ¿Y si deseamos que aparezcan exactamente 800 puntos, que es la anchura de la superficie de dibujo? Muy sencillo: podemos calcular el incremento dividiendo entre 800 el dominio de la función:

```
seno.py
from turtle import Screen, Turtle
from math import sin, pi

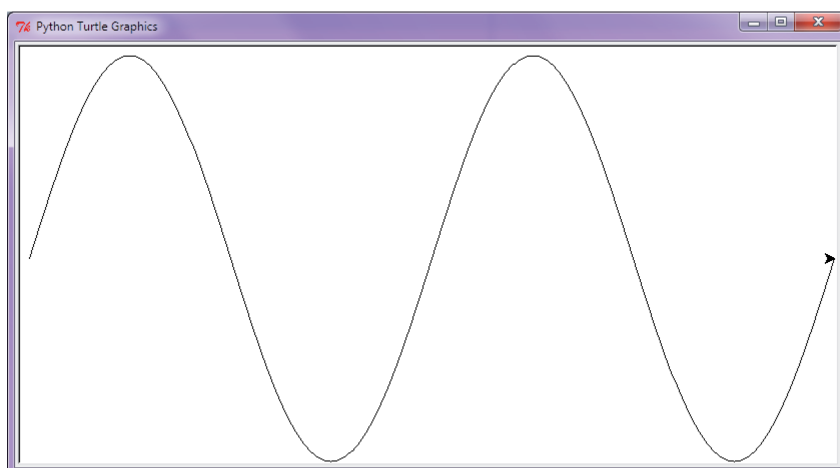
pantalla = Screen()
pantalla.setup(825, 425)
pantalla.screensize(800, 400)
pantalla.setworldcoordinates(-2*pi, -1, 2*pi, 1)

tortuga = Turtle()

x = -2*pi
dx = 4*pi / 800
tortuga.penup()
tortuga.goto(x, sin(x))
tortuga.pendown()
while x <= 2*pi:
    tortuga.goto(x, sin(x))
    x += dx

pantalla.exitonclick()
```

Este es el resultado:



Hagamos que el usuario pueda introducir el intervalo de valores de *x* que desea examinar, así como el número de puntos que desee representar:

```
seno.py
from turtle import Screen, Turtle
from math import sin, pi

x1 = float(input('Dime el límite inferior del intervalo: '))
x2 = float(input('Dime el límite superior del intervalo: '))
puntos = int(input('Dime cuántos puntos he de mostrar: '))

pantalla = Screen()
pantalla.setup(825, 425)
pantalla.screensize(800, 400)
pantalla.setworldcoordinates(x1, -1, x2, 1)

tortuga = Turtle()
```

```

x = x1
dx = (x2 - x1) / puntos
tortuga.penup()
tortuga.goto(x, sin(x))
tortuga.pendown()
while x <= x2:
    tortuga.goto(x, sin(x))
    x += dx

pantalla.exitonclick()
```

Prueba el programa con diferentes valores de las variables. Fíjate en qué programa tan útil hemos construido con muy pocos elementos: variables, bucles, el módulo *math* y unas pocas funciones predefinidas para trabajar con gráficos.

▶ 135 Haz un programa que muestre la función coseno en el intervalo que te indique el usuario.

▶ 136 Modifica el programa anterior para que se muestren dos funciones a la vez: la función seno y la función coseno, pero cada una en un color distinto.

▶ 137 Haz un programa que muestre la función $1/(x+1)$ en el intervalo $[-2, 2]$ con 100 puntos azules. Ten en cuenta que la función es «problemática» en $x = -1$, por lo que dibujaremos un punto rojo en las coordenadas $(-1, 0)$.

▶ 138 Haz un programa que, dados tres valores a, b y c, muestre la función $f(x) = ax^2 + bx+c$ en el intervalo $[z_1, z_2]$, donde z_1 y z_2 son valores proporcionados por el usuario. El programa de dibujo debe calcular el valor máximo y mínimo de $f(x)$ en el intervalo indicado para ajustar el valor de *pantalla.setworldcoordinates* de modo que la función se muestre sin recorte alguno.

▶ 139 Añade a la gráfica del ejercicio anterior una representación de los ejes coordenados en color azul. Dibuja con círculos rojos los puntos en los que la parábola $f(x)$ corta el eje horizontal. Recuerda que la parábola corta al eje horizontal en los puntos x_1 y x_2 que son solución de la ecuación de segundo grado $ax^2 + bx + c = 0$.

4.4.2. Una animación: simulación gravitacional

Vamos a construir ahora un pequeño programa de simulación gravitacional. Representaremos en pantalla dos cuerpos y veremos qué movimiento presentan bajo la influencia mutua de la gravedad en un universo bidimensional. Nos hará falta repasar algunas nociones básicas de física.

La ley de gravitación general de Newton nos dice que dos cuerpos de masas m_1 y m_2 se atraen con una fuerza

$$F = G\frac{m_1 m_2}{r^2},$$

donde G es la constante de gravitación universal y r es la distancia que separa a los cuerpos. Sometido a esa fuerza, cada cuerpo experimenta una aceleración. Recuerda que la aceleración a experimentada por un cuerpo de masa m sometido a una fuerza F es $a = F/m$. Cada cuerpo experimentará una aceleración distinta:

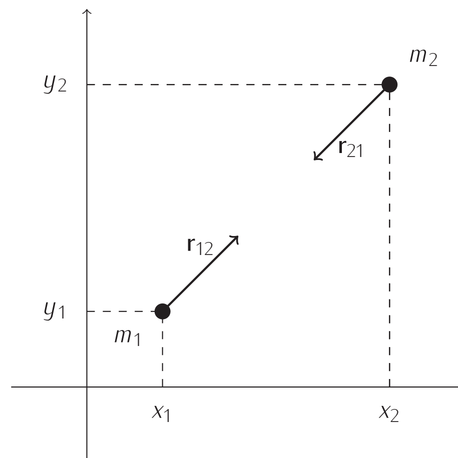
$$\begin{aligned} a_1 &= G\frac{m_2}{r^2}, \\ a_2 &= G\frac{m_1}{r^2}. \end{aligned}$$

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

Como los cuerpos ocupan las posiciones (x_1, y_1) y (x_2, y_2) en el plano, podemos dar una formulación vectorial de las fórmulas anteriores:

$$\mathbf{a}_1 = G\frac{m_2\mathbf{r}_{12}}{r^3},$$
$$\mathbf{a}_2 = G\frac{m_1\mathbf{r}_{21}}{r^3},$$

donde los símbolos en negrita son vectores.



En particular, $\mathbf{r}_{12}$ es el vector $(x_2 - x_1, y_2 - y_1)$ y $\mathbf{r}_{21}$ es el vector $(x_1 - x_2, y_1 - y_2)$. El valor de r, su módulo, es

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

La aceleración afecta en cada instante de tiempo a la velocidad de cada cuerpo. Si un cuerpo se desplaza en un instante dado a una velocidad (v_x, v_y), una unidad de tiempo más tarde se desplazará a velocidad $(v_x + a_x, v_y + a_y)$, siendo (a_x, a_y) su vector de aceleración. El vector de aceleración del primer cuerpo es proporcional a $\mathbf{r}_{12}$, y el del segundo cuerpo es proporcional a $\mathbf{r}_{21}$.

Ya basta de física. Volvamos al mundo de las tortugas. Representaremos cada cuerpo con una tortuga con la forma de un círculo. Al crear una tortuga con *Turtle* podemos proporcionar un parámetro de tipo cadena que indica la forma de la tortuga. Las formas disponibles son: flecha (`'arrow'`), nada (`'blank'`), círculo (`'circle'`), clásica (`'classic'`), cuadrado (`'square'`), triángulo (`'triangle'`) y tortuga (`'turtle'`). Por otra parte, podemos asignar un color a cada tortuga de modo que tanto la tortuga como su trazo se muestren en ese color. El método *color* admite una cadena con el color en cuestión (en inglés).

¿Con qué datos modelamos cada cuerpo? Una variable almacenará la masa de cada cuerpo, eso está claro. Llamemos a esas variables *m1* y *m2*. En cada instante, cada cuerpo ocupa una posición en el plano. Cada posición se representa con dos valores: la posición en el eje X y la posición en el eje Y. Las variables *x1* e *y1* almacenarán la posición del primer cuerpo y las variables *x2* e *y2* las del segundo. Otro dato importante es la velocidad que cada cuerpo lleva en un instante dado. La velocidad es un vector, así que necesitamos dos variables para representarla. Las variables *velocidad_x1* y *velocidad_y1* almacenarán el vector de velocidad del primer cuerpo y las variables *velocidad_x2* y *velocidad_y2* el del segundo. También la aceleración de cada cuerpo requiere dos variables y para representarla seguiremos el mismo patrón, solo que las variables empezarán con el prefijo *aceleración*.

Inicialmente cada cuerpo ocupa una posición y lleva una velocidad determinada. Nuestro programa puede empezar, de momento, así:

gravedad.py
```
x1 = -200
y1 = -200
velocidad_x1 = 0.1
```

```
velocidad_y1 = 0
m1 = 20

x2 = 200
y2 = 200
velocidad_x2 = -0.1
velocidad_y2 = 0
m2 = 20
```

Los cálculos que nos permiten actualizar los valores de posición y velocidad de cada cuerpo son, de acuerdo con las nociones de física que hemos repasado, estos:

```
gravedad.py
from math import sqrt

x1 = -200
y1 = -200
velocidad_x1 = 0.1
velocidad_y1 = 0
m1 = 20

x2 = 200
y2 = 200
velocidad_x2 = -0.1
velocidad_y2 = 0
m2 = 20

r = sqrt( (x2-x1)**2 + (y2-y1)**2 )

aceleración_x1 = m2 * (x2 - x1) / r**3
aceleración_y1 = m2 * (y2 - y1) / r**3
aceleración_x2 = m1 * (x1 - x2) / r**3
aceleración_y2 = m1 * (y1 - y2) / r**3

velocidad_x1 += aceleración_x1
velocidad_y1 += aceleración_y1
velocidad_x2 += aceleración_x2
velocidad_y2 += aceleración_y2

x1 += velocidad_x1
y1 += velocidad_y1
x2 += velocidad_x2
y2 += velocidad_y2
```

Advertirás que no hemos usado la constante de gravitación G. Como afecta linealmente a la fórmula, su único efecto práctico es «acelerar» la simulación, así que hemos decidido prescindir de ella.

Creemos la pantalla y hagamos que cada cuerpo se represente con una tortuga de un color diferente:

```
gravedad.py
from turtle import Screen, Turtle
from math import sqrt

pantalla = Screen()
pantalla.setup(1025, 1025)
pantalla.screensize(1000, 1000)
pantalla.setworldcoordinates(-500, -500, 500, 500)

x1 = -200
y1 = -200
```

```
velocidad_x1 = 0.1
velocidad_y1 = 0
m1 = 20

x2 = 200
y2 = 200
velocidad_x2 = -0.1
velocidad_y2 = 0
m2 = 20

cuerpo1 = Turtle('circle')
cuerpo1.color('red')
cuerpo1.speed(0)
cuerpo1.penup()
cuerpo1.goto(x1, y1)
cuerpo1.pendown()

cuerpo2 = Turtle('circle')
cuerpo2.color('blue')
cuerpo2.speed(0)
cuerpo2.penup()
cuerpo2.goto(x2, y2)
cuerpo2.pendown()

r = sqrt( (x2-x1)**2 + (y2-y1)**2 )

aceleración_x1 = m2 * (x2 - x1) / r**3
aceleración_y1 = m2 * (y2 - y1) / r**3
aceleración_x2 = m1 * (x1 - x2) / r**3
aceleración_y2 = m1 * (y1 - y2) / r**3

velocidad_x1 += aceleración_x1
velocidad_y1 += aceleración_y1
velocidad_x2 += aceleración_x2
velocidad_y2 += aceleración_y2

x1 += velocidad_x1
y1 += velocidad_y1
x2 += velocidad_x2
y2 += velocidad_y2

cuerpo1.goto(x1, y1)
cuerpo2.goto(x2, y2)

pantalla.exitonclick()
```

Hemos aprovechado para fijar la velocidad de las dos tortugas al máximo y para, tras ubicarlas en sus respectivos puntos de partida, trasladarlas a la posición que ocupan un instante de tiempo después.

Si queremos ver cómo evolucionan los cuerpos a lo largo del tiempo, deberemos repetir este cálculo numerosas veces, así que formará parte de un bucle. Para ver qué ocurre a lo largo de 10.000 unidades de tiempo, por ejemplo, insertaremos esa serie de acciones en un bucle al final del cual se redibujan los dos cuerpos:

gravedad.py

```
from turtle import Screen, Turtle
from math import sqrt

pantalla = Screen()
pantalla.setup(1025, 1025)
pantalla.screensize(1000, 1000)
```

```
pantalla.setworldcoordinates(-500, -500, 500, 500)

x1 = -200
y1 = -200
velocidad_x1 = 0.1
velocidad_y1 = 0
m1 = 20

x2 = 200
y2 = 200
velocidad_x2 = -0.1
velocidad_y2 = 0
m2 = 20

cuerpo1 = Turtle('circle')
cuerpo1.color('red')
cuerpo1.speed(0)
cuerpo1.penup()
cuerpo1.goto(x1, y1)
cuerpo1.pendown()

cuerpo2 = Turtle('circle')
cuerpo2.color('blue')
cuerpo2.speed(0)
cuerpo2.penup()
cuerpo2.goto(x2, y2)
cuerpo2.pendown()

for t in range(10000):
    r = sqrt( (x2-x1)**2 + (y2-y1)**2 )

    aceleración_x1 = m2 * (x2 - x1) / r**3
    aceleración_y1 = m2 * (y2 - y1) / r**3
    aceleración_x2 = m1 * (x1 - x2) / r**3
    aceleración_y2 = m1 * (y1 - y2) / r**3

    velocidad_x1 += aceleración_x1
    velocidad_y1 += aceleración_y1
    velocidad_x2 += aceleración_x2
    velocidad_y2 += aceleración_y2

    x1 += velocidad_x1
    y1 += velocidad_y1
    x2 += velocidad_x2
    y2 += velocidad_y2

    cuerpo1.goto(x1, y1)
    cuerpo2.goto(x2, y2)

pantalla.exitonclick()
```

Hay un escollo que salvar: la animación es muy lenta. Es un efecto buscado por el diseñador del módulo *turtle*, pues al ser un módulo diseñado con intención pedagógica, las tortugas se mueven lentamente y así resulta más fácil percibir qué ocurre durante la ejecución del programa (especialmente si algo va mal). Pero el diseñador dejó una puerta abierta: con el método *delay* podemos fijar el tiempo que transcurre entre dos «fotogramas» de la animación.

gravedad.py

```
from turtle import Screen, Turtle
from math import sqrt

```

```
pantalla = Screen()
pantalla.setup(1025, 1025)
pantalla.screensize(1000, 1000)
pantalla.setworldcoordinates(-500, -500, 500, 500)
pantalla.delay(0)

x1 = -200
y1 = -200
velocidad_x1 = 0.1
velocidad_y1 = 0
m1 = 20

x2 = 200
y2 = 200
velocidad_x2 = -0.1
velocidad_y2 = 0
m2 = 20

cuerpo1 = Turtle('circle')
cuerpo1.color('red')
cuerpo1.speed(0)
cuerpo1.penup()
cuerpo1.goto(x1, y1)
cuerpo1.pendown()

cuerpo2 = Turtle('circle')
cuerpo2.color('blue')
cuerpo2.speed(0)
cuerpo2.penup()
cuerpo2.goto(x2, y2)
cuerpo2.pendown()

for t in range(10000):
    r = sqrt( (x2-x1)**2 + (y2-y1)**2 )

    aceleración_x1 = m2 * (x2 - x1) / r**3
    aceleración_y1 = m2 * (y2 - y1) / r**3
    aceleración_x2 = m1 * (x1 - x2) / r**3
    aceleración_y2 = m1 * (y1 - y2) / r**3

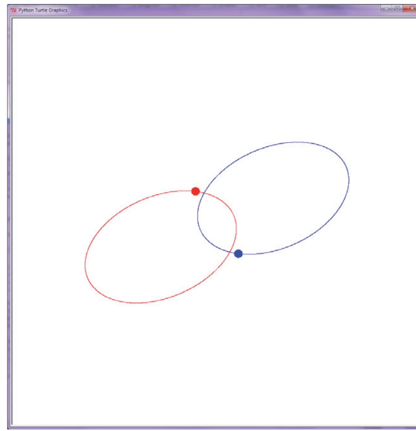
    velocidad_x1 += aceleración_x1
    velocidad_y1 += aceleración_y1
    velocidad_x2 += aceleración_x2
    velocidad_y2 += aceleración_y2

    x1 += velocidad_x1
    y1 += velocidad_y1
    x2 += velocidad_x2
    y2 += velocidad_y2

    cuerpo1.goto(x1, y1)
    cuerpo2.goto(x2, y2)

pantalla.exitonclick()
```

Y ya está: ejecutemos el programa. He aquí el resultado final (en pantalla aparecerá como una animación):



Es momento de alguna optimización. Elevar un número al cubo es una operación costosa. Nosotros la efectuamos cuatro veces cuando una sola es suficiente. Deberíamos calcular r^3 una sola vez, almacenar el resultado en una variable y usar esa variable como divisor en las cuatro divisiones que se efectúan en cada bucle. Dejamos esta optimización como ejercicio para el lector.

Diviértete con el programa. He aquí algunas configuraciones iniciales interesantes:

1)

```
x1 = -200
y1 = -200
velocidad_x1 = 0.1
velocidad_y1 = 0
m1 = 0.001

x2 = 200
y2 = 200
velocidad_x2 = 0
velocidad_y2 = 0
m2 = 20
```

2)

```
x1 = -200
y1 = -200
velocidad_x1 = -0.1
velocidad_y1 = 0
m1 = 20

x2 = 200
y2 = 200
velocidad_x2 = -0.1
velocidad_y2 = 0
m2 = 20
```

▶ 140 ¿Qué pasaría si los dos cuerpos ocuparan exactamente la misma posición en el plano? Modifica el programa para que, si se da el caso, no se produzca error alguno y finalice inmediatamente la ejecución del bucle.

▶ 141 Modifica el programa para que la simulación no finalice nunca (bueno, solo cuando el usuario interrumpa la ejecución del programa).

▶ 142 ¿Serías capaz de extender el programa para que muestre la interacción entre tres cuerpos? Repasa la formulación física del problema antes de empezar a programar.

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

4.5. Una reflexión final

En este capítulo te hemos presentado varias estructuras de control de flujo que, esencialmente, se reducen a dos conceptos: la *selección condicional* de sentencias y la *repetición condicional* de sentencias. En los primeros tiempos de la programación no siempre se utilizaban estas estructuras: existía una sentencia comodín que permitía «saltar» a cualquier punto de un programa: la que se conoce como sentencia `goto` (en inglés, «ir-a»).

Observa cómo se podría haber escrito el programa `es_primo.py` (sección 4.2.7) en el lenguaje de programación BASIC, que originariamente carecía de estructuras como el bucle **while**:

```
 10 INPUT "DAME UN NÚMERO:"; NUM
 20 DIVISOR = 2
 30 IF INT(NUM / DIVISOR) = NUM / DIVISOR THEN GOTO 90
 40 DIVISOR = DIVISOR + 1
 50 IF DIVISOR = NUM THEN GOTO 70
 60 GOTO 30
 70 PRINT . El número", NUM, " es primo"
 80 GOTO 100
 90 PRINT "El número", NUM, "no es primo"
100 END
```

Cada línea del programa está numerada y la sentencia `GOTO` indica en qué línea debe continuar la ejecución del programa. Como es posible saltar a cualquier línea en función de la satisfacción de una condición, es posible «montar a mano» cualquier estructura de control. Ahora bien, una cosa es que sea posible y otra que el resultado presente un mínimo de elegancia. El programa BASIC del ejemplo es endiabladamente complejo: resulta difícil apreciar que las líneas `30`–`60` forman un bucle **while**. Los programas construidos abusando del `GOTO` recibían el nombre de «código spaghetti», pues al representar con flechas los posibles saltos del programa se formaba una maraña que recuerda a un plato de *spaghetti*.

En los años 70 hubo una corriente en el campo de la informática que propugnaba la supresión de la sentencia `goto`. Edsger W. Dijkstra publicó un influyente artículo titulado «Goto considered harmful» («La sentencia `Goto` considerada dañina») en el que se hacía una severa crítica al uso de esta sentencia en los programas. Se demostró que era posible construir cualquier programa con solo selecciones y repeticiones condicionales y que estos programas resultaban mucho más legibles. La denominada *programación estructurada* es la corriente que propugna (entre otros principios) la programación usando únicamente *estructuras de control* (**if**, **while**, **for-in**,...) para alterar el flujo del programa.

Al poco tiempo de su aparición, la programación estructurada se convirtió en *la* metodología de programación. (Los puristas de la programación estructurada no solo censuran el uso de sentencias `goto`: también otras como **break** están proscritas).

Hay que decir, no obstante, que programar es una forma de describir ideas algorítmicas siguiendo unas reglas sintácticas determinadas y que, en ocasiones, romper una regla permite una mejor expresión. Pero, ¡ojo!, solo estarás capacitado para romper reglas cuando las conozcas *perfectamente*. Por una cuestión de disciplina es preferible que, al principio, procures no utilizar en absoluto alteraciones del flujo de control arbitrarias... aunque de todos modos no podrás hacerlo de momento: ¡Python no tiene sentencia `goto`!

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

Capítulo 5

Tipos estructurados: secuencias

> Primero llegaron diez soldados portando bastos: tenían la misma forma que los tres jardineros, plana y rectangular, con las manos y los pies en las esquinas; luego venían los diez cortesanos, todos adornados de diamantes, y caminaban de dos en dos, como los soldados. Seguían los infantes: eran diez en total y era encantador verlos venir cogidos de la mano, en parejas, dando alegres saltos: estaban adornados con corazones.
>
> *Alicia en el país de las maravillas,* Lewis Carroll

Hasta el momento hemos tratado con datos de cuatro tipos distintos: enteros, flotantes, lógicos y cadenas. Los tres primeros son tipos de datos *escalares*. Las cadenas, por contra, son tipos de datos *secuenciales*. Un dato de tipo escalar es un elemento único, atómico. Por contra, un dato de tipo secuencial se compone de una *sucesión* de elementos y una cadena es una sucesión de caracteres. Los datos de tipo secuencial son *datos estructurados*. En Python es posible manipular los datos secuenciales de diferentes modos, facilitando así la escritura de programas que manejan conjuntos o series de valores.

En algunos puntos de la exposición nos desviaremos hacia cuestiones relativas al modelo de memoria de Python. Aunque se trata de un material que debes comprender y dominar, no pierdas de vista que lo realmente importante es que aprendas a diseñar e implementar algoritmos que trabajan con secuencias.

En este capítulo empezaremos aprendiendo más de lo que ya sabemos sobre *cadenas*. Después, te presentaremos las *listas*. Una lista es una sucesión de elementos de cualquier tipo. Finalmente, aprenderás a definir y manejar *matrices*: disposiciones bidimensionales de elementos. Python no incorpora un tipo de datos nativo para matrices, así que las construiremos como *listas de listas*.

5.1. Cadenas

5.1.1. Lo que ya sabemos

Ya vimos en capítulos anteriores que una *cadena* es una sucesión de caracteres. Python ofrece una serie de operadores y funciones predefinidos que manipulan cadenas o devuelven cadenas como resultado. Repasemos brevemente las que ya conocemos de capítulos anteriores:

- Operador + (concatenación de cadenas): acepta dos cadenas como operandos y devuelve la cadena que resulta de unir la segunda a la primera.

- Operador * (repetición de cadena): acepta una cadena y un entero y devuelve la concatenación de la cadena consigo misma tantas veces como indica el entero.

- *int*: recibe una cadena cuyo contenido es una secuencia de dígitos y devuelve el número entero que describe.

- *float*: acepta una cadena cuyo contenido describe un flotante y devuelve el flotante en cuestión.
- *str*: se le pasa un entero o flotante y devuelve una cadena con una representación del valor como secuencia de caracteres.
- *ord*: acepta una cadena compuesta por un único carácter y devuelve su código Unicode (un entero).
- *chr*: recibe un entero y devuelve una cadena con el carácter que tiene a dicho entero como código Unicode.

Podemos manipular cadenas, además, mediante métodos que les son propios:

- *a*.*lower*() (paso a minúsculas): devuelve una cadena con los caracteres de *a* convertidos en minúsculas.
- *a*.*upper*() (paso a mayúsculas): devuelve una cadena con los caracteres de *a* convertidos en mayúsculas.
- *a*.*title*() (paso a palabras con inicial mayúscula): devuelve una cadena en la que toda palabra de *a* empieza por mayúscula.
- *a*.*format*(*expr1*, *expr2*, ...) (sustitución de marcas de formato): devuelve una cadena en la que las marcas de formato de *a* se sustituyen por el resultado de evaluar las expresiones dadas.

Aprenderemos ahora a utilizar nuevas herramientas. Pero antes, estudiemos algunas peculiaridades de la codificación de los caracteres en las cadenas.

5.1.2. Escapes

Las cadenas que hemos estudiado hasta el momento consistían en sucesiones de caracteres «normales»: letras, dígitos, signos de puntuación, espacios en blanco... Es posible, no obstante, incluir ciertos caracteres especiales que no tienen una representación trivial.

Por ejemplo, los *saltos de línea* se muestran en pantalla como eso, saltos de línea, no como un carácter convencional. Si intentamos incluir un salto de línea en una cadena pulsando la tecla de retorno de carro, Python se queja:

```
>>> a = 'una↵
  File "<input>", line 1
    a = 'una
        ^
SyntaxError: EOL while scanning string literal
```

¿Ves? Al pulsar la tecla de retorno de carro, el intérprete de Python intenta ejecutar la sentencia inmediatamente y considera que la cadena está inacabada, así que notifica que ha detectado un error.

Observa esta otra asignación de una cadena a la variable *a* y mira qué ocurre cuando mostramos el contenido de *a*:

```
>>> a = 'una\ncadena'↵
>>> print(a)↵
una
cadena
```

Al mostrar la cadena se ha producido un salto de línea detrás de la palabra *una*. El salto de línea se ha codificado en la cadena con dos caracteres: la barra invertida \ y la letra *n*.

La barra invertida se denomina *carácter de escape* y es un carácter especial: indica que el siguiente carácter tiene un significado diferente del usual. Si el carácter que le sigue es la letra *n*, por ejemplo, se interpreta como un salto de línea (la *n* viene del término «new line», es decir, «nueva línea»). Ese par de caracteres forma una *secuencia de escape* y denota un único carácter. ¿Y un salto de línea es un único carácter? Sí. Ocupa el mismo espacio en memoria que cualquier otro carácter y se codifica internamente con un valor numérico (código Unicode): el valor 10.

```
>>> ord('\n')↵
10
```

Cuando una impresora o un terminal de pantalla tratan de representar el carácter de código 10, saltan de línea. El carácter \n es un carácter *de control*, pues su función es permitirnos ejecutar una acción de control sobre ciertos dispositivos (como la impresora o el terminal).

Mostrar con y sin *print*

Es un buen momento para que veas la diferencia que supone usar o no usar la función *print* en el intérprete interactivo a la hora de mostrar el valor de una variable:

```
>>> a = 'una\ncadena'↵
>>> a↵
'una\ncadena'
>>> print(a)↵
una
cadena
```

¿Ves? Al mostrar directamente el valor de *a*, lo vemos como una cadena Python y, en consecuencia, el salto de línea aparece representado con `\n`. Por el contrario, si lo mostramos con *print*, se imprime en pantalla cada elemento de la cadena, teniendo en cuenta que las comillas no son elementos de la cadena (sino las marcas que usa Python para saber o señalar dónde empieza y acaba esta) y que el carácter de salto de línea provoca en pantalla un salto de línea.

Hay muchos caracteres de control. Esta tabla muestra algunos:

Secuencia de escape para carácter de control	Resultado
`\a`	Carácter de «campana» (`BEL`)
`\b`	«Espacio atrás» (`BS`)
`\f`	Alimentación de formulario (`FF`)
`\n`	Salto de línea (`LF`)
`\r`	Retorno de carro (`CR`)
`\t`	Tabulador horizontal (`TAB`)
`\v`	Tabulador vertical (`VT`)
`\`*ooo*	Carácter cuyo código en octal es *ooo*
`\x`*hh*	Carácter cuyo código en hexadecimal es *hh*

Pero no te preocupes: nosotros utilizaremos fundamentalmente dos: \n y \t. Este último representa el carácter de tabulación horizontal o, simplemente, tabulador. El tabulador puede resultar útil para alinear en columnas datos mostrados por pantalla. Mira este ejemplo, en el que destacamos los espacios en blanco de la salida por pantalla para que puedas contarlos:

```
>>> print('uno\tdos\ttres')↵
uno␣␣␣␣␣dos␣␣␣␣␣tres
>>> print('1\t2\t3')↵
1␣␣␣␣␣␣␣2␣␣␣␣␣␣␣3
>>> print('aa\tbb\tcc\nxx\tyy\tzz')↵
aa␣␣␣␣␣␣bb␣␣␣␣␣␣cc
xx␣␣␣␣␣␣yy␣␣␣␣␣␣zz
```

Los elementos se alinean en la misma columna porque hay marcas de alineación cada 8 columnas. El tabulador se interpreta como «desplázate a la siguiente marca de alineación».

Alternativamente, puedes usar el código (en octal o hexadecimal) de un carácter de control para codificarlo en una cadena, como se muestra en las dos últimas filas de la tabla de secuencias de escape. El salto de línea tiene código 10, que en octal se codifica con `\012` y en hexadecimal con `\x0a`. Aquí te mostramos una cadena con tres saltos de línea codificados de diferente forma:

```
>>> print('A\nB\012C\x0aD')↵
A
B
C
D
```

Ciertos caracteres no se pueden representar directamente en una cadena. La barra invertida es uno de ellos. Para expresarla, debes usar dos barras invertidas seguidas:

```
>>> print('a\\b')↵
a\b
```

En una cadena delimitada con comillas simples no puedes usar una comilla simple: si Python trata de analizar una cadena mal formada, como `'Munich'72'`, encuentra un error, pues cree que la cadena es `'Munich'` y no sabe cómo interpretar los caracteres `72'`. Una comilla simple en una cadena delimitada con comillas simples ha de ir precedida de la barra invertida. Lo mismo ocurre con la comilla doble en una cadena delimitada con comillas dobles:

```
>>> print('Munich\'72')↵
Munich'72
>>> print("Una␣\"cosa\"␣rara.")↵
Una "cosa" rara.
```

Estas secuencias de escape se pueden evitar la mayor parte de las veces escogiendo apropiadamente las comillas simples o dobles como delimitadores de la cadena:

```
>>> print("Munich'72")↵
Munich'72
>>> print('Una␣"cosa"␣rara.')↵
Una "cosa" rara.
```

Esta tabla complementa a la última:

Otras secuencias de escape	Resultado
`\\`	Carácter barra invertida (\)
`\'`	Comilla simple (')
`\"`	Comilla doble (")
\ y salto de línea	Se ignora (para expresar una cadena en varias líneas)

Fíjate en el uso especial de la barra invertida al preceder a un carácter de salto de línea:

```
>>> print('a\↵
... b')↵
ab
```

Unix, Microsoft y Apple: condenados a no entenderse

Te hemos dicho que `\n` codifica el carácter de control «salto de línea». Es cierto, pero no es toda la verdad. En los antiquísimos sistemas de teletipo (básicamente, máquinas de escribir controladas por ordenador que se usaban antes de que existieran los monitores) se necesitaban dos caracteres para empezar a escribir al principio de la siguiente línea: un salto de línea (`\n`) y un retorno de carro (`\r`). Si solo se enviaba el carácter `\n` el «carro» saltaba a la siguiente línea, sí, pero se quedaba en la misma columna. El carácter `\r` hacía que el carro retornase a la primera columna.

Con objeto de ahorrar memoria, los diseñadores de Unix decidieron que el final de línea en un fichero debería marcarse únicamente con `\n`. Al diseñar MS-DOS, Microsoft optó por utilizar dos caracteres: `\n\r`. Así pues, los ficheros de texto de Unix no son directamente compatibles con los de Microsoft. Si llevas un fichero de texto de un sistema Microsoft a Unix verás que cada línea acaba con un símbolo extraño (¡el retorno de carro!), y si llevas el fichero de Unix a un sistema Microsoft, parecerá que las líneas están mal alineadas.

Para poner peor las cosas, nos falta hablar de la decisión que adoptó Apple en los ordenadores Macintosh: usar solo el retorno de carro (`\r`). ¡Tres sistemas operativos y tres formas distintas de decir lo mismo!

De todos modos, no te preocupes en exceso, muchos editores de texto son suficientemente «listos»: pueden detectar estas situaciones y las corrigen automáticamente. En otros, el usuario puede ir a un panel de preferencias y seleccionar el modo con el que se representan internamente los saltos de línea.

▶ 143 ¿Qué se mostrará en pantalla al ejecutar estas sentencias?

```
>>> print('\\n')↵
>>> print('\157\143\164\141\154')↵
>>> print('\t\tuna\bo')↵
```

(Te recomendamos que resuelvas este ejercicio a mano y compruebes la validez de tus respuestas con ayuda del ordenador).

▶ 144 ¿Cómo crees que se pueden representar dos barras invertidas seguidas en una cadena?

▶ 145 La secuencia de escape `\a` emite un aviso sonoro (la «campana»). ¿Qué hace exactamente cuando se imprime en pantalla? Ejecuta *print*`('\a')` y lo averiguarás.

▶ 146 Averigua el código Unicode de los 10 primeros caracteres de la tabla en la que hemos mostrado las secuencias de escape para caracteres de control.

Más sobre la codificación de las cadenas

Hemos visto que podemos codificar cadenas encerrando un texto entre comillas simples o entre comillas dobles. En tal caso, necesitamos usar secuencias de escape para acceder a ciertos caracteres. Python ofrece aún más posibilidades para codificar cadenas. Una de ellas hace que no se interpreten las secuencias de escape, es decir, que todos sus caracteres se interpreten literalmente. Estas cadenas «directas» (en inglés, «raw strings») preceden con la letra «r» a las comillas (simples o dobles) que la inician:

```
>>> print(r'u\n')↵
u\n
>>> print("u\\n")↵
u\n
```

Cuando una cadena ocupa varias líneas, podemos usar la secuencia de escape `\n` para marcar cada salto de línea. O podemos usar una «cadena multilínea». Las cadenas multilínea empiezan con tres comillas simples (o dobles) y finalizan con tres comillas simples (o dobles):

```
>>> print('''Una↵
...  cadena↵
...  que  ocupa↵
...  varias  líneas''')↵
Una
cadena
que  ocupa
varias  líneas
```

5.1.3. Longitud de una cadena

La primera nueva función que estudiaremos es *len* (abreviatura del inglés «length», en español, «longitud») que devuelve la longitud de una cadena, es decir, el número de caracteres que la forman. Se trata de una función predefinida, así que podemos usarla directamente:

```
>>> len('abc') ↵
3
>>> len('a')↵
1
>>> len('abcd' * 4)↵
16
>>> len('a\nb')↵
3
```

Hay una cadena que merece especial atención, la cadena que denotamos abriendo y cerrando inmediatamente las comillas simples, `''`, o dobles, `""`, sin ningún carácter entre ellas. ¿Qué valor devuelve *len*`('')`?

```
>>> len('')↵
0
```

La cadena `''` se denomina *cadena vacía* y tiene longitud cero. No confundas la cadena vacía, `''`, con la cadena que contiene un espacio en blanco, `'␣'`, pues, aunque parecidas, no son iguales. Fíjate bien en que la segunda cadena contiene un carácter (el espacio en blanco) y, por tanto, es de longitud 1. Podemos comprobarlo fácilmente:

```
>>> len('')↵
0
>>> len('␣')↵
1
```

5.1.4. Indexación

Podemos acceder a cada uno de los caracteres de una cadena utilizando un operador de *indexación*. El índice del elemento al que queremos acceder debe encerrarse entre corchetes. Si *a* es una cadena, *a*[*i*] es el carácter que ocupa la posición *i*+1. Debes tener en cuenta que el primer elemento tiene índice cero. Los índices de la cadena `'Hola,␣mundo.'` se muestran en esta figura:

0	1	2	3	4	5	6	7	8	9	10	11
H	o	l	a	,	␣	m	u	n	d	o	.

```
>>> 'Hola,␣mundo.'[0]↵
'H'
>>> 'Hola,␣mundo.'[1]↵
'o'
>>> a = 'Hola,␣mundo.'↵
>>> a[2]↵
'l'
>>> a[1]↵
'o'
>>> i = 3↵
>>> a[i]↵
'a'
>>> a[len(a)-1]↵
'.'
```

Observa que el último carácter de la cadena almacenada en la variable *a* no es *a*[*len*(*a*)], sino *a*[*len*(*a*)-1]. ¿Por qué? Evidentemente, si el primer carácter tiene índice 0 y hay *len*(*a*) caracteres, el último ha de tener índice *len*(*a*)-1. Si intentamos acceder al elemento *a*[*len*(*a*)], Python protesta:

```
>>> a = "cadena"↵
>>> a[len(a)]↵
Traceback (most recent call last):
  File "<input>", line 1, in <module>
IndexError: string index out of range
```

El error cometido es del tipo *IndexError* (error de indexación) y, en el texto explicativo que lo detalla, Python nos informa de que el índice de la cadena está fuera del rango de valores válidos.

Recuerda que las secuencias de escape codifican caracteres simples, aunque se expresen con dos caracteres. La cadena `'Hola,\nmundo.'`, por ejemplo, no ocupa 13 casillas, sino 12:

0	1	2	3	4	5	6	7	8	9	10	11
H	o	l	a	,	\n	m	u	n	d	o	.

También puedes utilizar índices negativos con un significado especial: los valores negativos acceden a los caracteres de derecha a izquierda. El último carácter de una cadena tiene índice −1, el penúltimo, −2, y así sucesivamente. Analiza este ejemplo:

```
>>> a = 'Ejemplo'↵
>>> a[-1]↵
'o'
>>> a[len(a)-1]↵
```

```
'o'
>>> a[-3]↵
'p'
>>> a[-len(a)]↵
'E'
```

De este modo se simplifica notablemente el acceso a los caracteres del final de la cadena. Es como si dispusieras de un doble juego de índices:

▶ 147 La última letra del DNI puede calcularse a partir de sus números. Para ello solo tienes que dividir el número por 23 y quedarte con el resto. El resto es un número entre 0 y 22. La letra que corresponde a cada número la tienes en esta tabla:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
T	R	W	A	G	M	Y	F	P	D	X	B	N	J	Z	S	Q	V	H	L	C	K	E

Diseña un programa que lea de teclado un número de DNI y muestre en pantalla la letra que le corresponde.

(Nota: una implementación basada en tomar una decisión con **if-elif** conduce a un programa muy largo. Si usas el operador de indexación de cadenas de forma inteligente, el programa apenas ocupa tres líneas. Piensa cómo).

5.1.5. Recorrido de cadenas

Una propiedad interesante de los datos secuenciales es que pueden recorrerse de izquierda a derecha con un bucle **for-in**. Por ejemplo, el siguiente bucle recorre los caracteres de una cadena de uno en uno, de izquierda a derecha:

```
>>> for carácter in "mi␣cadena":↵
...     print(carácter)↵
... ↵
m
i

c
a
d
e
n
a
```

En cada paso, la variable del bucle (en el ejemplo, *carácter*) toma el valor de uno de los caracteres de la cadena. Es lo que cabía esperar: recuerda que el bucle **for-in** recorre uno a uno los elementos de una serie de valores, y una cadena es una secuencia de caracteres.

Tienes una forma alternativa de recorrer los elementos de una cadena: recorriendo el rango de valores que toma su índice e indexando cada uno de ellos. Estudia este ejemplo:

```
>>> a = "mi␣cadena"↵
>>> for i in range(len(a)):↵
...     print(a[i])↵
... ↵
m
i

c
a
d
e
n
a
```

La variable i toma los valores de $range(len(a))$, en este caso los valores comprendidos entre 0 y 8, ambos inclusive. Con $a[i]$ hemos accedido, pues, a cada uno de ellos. Si mostramos tanto i como $a[i]$, quizás entiendas mejor qué ocurre exactamente:

```
>>> a = "mi␣cadena"↵
>>> for i in range(len(a)):↵
...     print(i, a[i])↵
... ↵
0 m
1 i
2
3 c
4 a
5 d
6 e
7 n
8 a
```

También puedes mostrar los caracteres de la cadena en orden inverso, aunque en tal caso has de hacerlo necesariamente con un bucle **for-in** y un *range*:

```
>>> a = "mi␣cadena"↵
>>> for i in range(len(a)):↵
...     print(a[len(a)-i-1])↵
... ↵
a
n
e
d
a
c

i
m
```

▶ 148 Intentamos mostrar los caracteres de la cadena en orden inverso así:

```
>>> a = "mi␣cadena"↵
>>> for i in range(len(a), -1):↵
...     print(a[i])↵
... ↵
```

¿Funciona?

▶ 149 Intentamos mostrar los caracteres de la cadena en orden inverso así:

```
>>> a = "mi␣cadena"↵
>>> for i in range(len(a)-1, -1, -1):↵
...     print(a[i])↵
... ↵
```

¿Funciona?

▶ 150 Diseña un programa que lea una cadena y muestre el número de espacios en blanco que contiene.

▶ 151 Diseña un programa que lea una cadena y muestre el número de letras mayúsculas que contiene.

▶ 152 Diseña un programa que lea una cadena y muestre en pantalla el mensaje «`Contiene dígito`» si contiene algún dígito y «`No contiene dígito`» en caso contrario.

5.1.6. Un ejemplo: un contador de palabras

Ahora que tenemos nuevas herramientas para la manipulación de cadenas, vamos a desarrollar un programa interesante: leerá cadenas de teclado y mostrará en pantalla el número de palabras que contienen.

Empecemos estudiando el problema con un ejemplo concreto. ¿Cuántas palabras hay en la cadena `'una␣dos␣tres'`? Tres palabras. ¿Cómo lo sabemos? Muy fácil: contando los espacios en blanco. Si hay *dos* espacios en blanco, entonces hay *tres* palabras, ya que cada espacio separa dos palabras. Hagamos, pues, que el programa cuente el número de espacios en blanco y muestre ese número más uno:

palabras.py
```
cadena = input('Escribe␣una␣frase:␣')
while cadena != '':
    blancos = 0
    for carácter in cadena:
        if carácter == '␣':
            blancos += 1
    palabras = blancos + 1 # Hay una palabra más que blancos
    print('Palabras:', palabras)

    cadena = input('Escribe␣una␣frase:␣')
```

El programa finaliza la ejecución cuando tecleamos una cadena vacía, es decir, si pulsamos retorno de carro directamente. Ejecutemos el programa:

```
Escribe una frase: una␣dos␣tres↵
Palabras: 3
Escribe una frase: mi␣ejemplo↵
Palabras: 2
Escribe una frase: ejemplo↵
Palabras: 1
Escribe una frase: otro␣␣ejemplo↵
Palabras: 3
Escribe una frase: ↵
```

¡Eh! ¿Qué ha pasado con el último ejemplo? Hay dos palabras y el programa dice que hay tres. Está claro: entre las palabras «otro» y «ejemplo» de la cadena `'otro␣␣ejemplo'` hay *dos* espacios en blanco, y no uno solo. Corrijamos el programa para que trate correctamente casos como este. Desde luego, contar espacios en blanco, sin más, no es la clave para decidir cuántas palabras hay. Se nos ocurre una idea mejor: mientras recorremos la cadena, veamos cuántas veces pasamos de un carácter que no sea el espacio en blanco a un espacio en blanco. En la cadena `'una␣dos␣tres'` pasamos *dos* veces de letra a espacio en blanco (una vez pasamos de la «`a`» al blanco y otra de la «`s`» al blanco), y hay *tres* palabras; en la cadena problemática `'otro␣␣ejemplo'` solo pasamos *una* vez (de la letra «`o`» a un espacio en blanco) y, por tanto, hay *dos* palabras. Si contamos el número de transiciones, el número de palabras será ese mismo número más uno. ¿Y cómo hacemos para comparar un carácter y su vecino? El truco está en recordar siempre cuál era el carácter anterior usando una variable auxiliar:

palabras.py
```
cadena = input('Escribe␣una␣frase:␣')
while cadena != '':
    cambios = 0
    anterior = ''
    for carácter in cadena:
        if carácter == '␣' and anterior != '␣':
            cambios += 1
        anterior = carácter
    palabras = cambios + 1 # Hay una palabra más que cambios de no blanco a blanco
    print('Palabras:', palabras)

    cadena = input('Escribe␣una␣frase:␣')
```

¿Por qué hemos dado un valor a *anterior* en la línea 4? Para inicializar la variable. De no hacerlo, tendríamos problemas al ejecutar la línea 6 por primera vez, ya que en ella se consulta el valor de *anterior*.

▶ 153 Haz una traza del programa para la cadena `'a␣b'`. ¿Qué líneas se ejecutan y qué valores toman las variables *cambios*, *anterior* y *carácter* tras la ejecución de cada una de ellas?

▶ 154 Ídem para la cadena `'a␣␣b'`.

Probemos nuestra nueva versión:

```
Escribe una frase: una␣dos␣tres↵
Palabras: 3
Escribe una frase: mi␣ejemplo↵
Palabras: 2
Escribe una frase: ejemplo↵
Palabras: 1
Escribe una frase: otro␣␣ejemplo␣↵
Palabras: 3
Escribe una frase: ↵
```

¡No! ¡Otra vez mal! ¿Qué ha ocurrido ahora? Si nos fijamos bien veremos que la cadena del último ejemplo acaba en un espacio en blanco, así que hay una transición de «no blanco» a espacio en blanco y eso, para nuestro programa, significa que hay una nueva palabra. ¿Cómo podemos corregir ese problema? Analicémoslo: parece que solo nos molestan los blancos *al final* de la cadena. ¿Y si descontamos una palabra cuando la cadena acaba en un espacio en blanco?

palabras.py
```
1  cadena = input('Escribe␣una␣frase:␣')
2  while cadena != '':
3      cambios = 0
4      anterior = ''
5      for carácter in cadena:
6          if carácter == '␣' and anterior != '␣':
7              cambios += 1
8          anterior = carácter
9
10     if cadena[-1] == '␣':
11         cambios -= 1
12
13     palabras = cambios + 1
14     print('Palabras:', palabras)
15
16     cadena = input('Escribe␣una␣frase:␣')
```

Probemos ahora esta nueva versión:

```
Escribe una frase: una␣dos␣tres↵
Palabras: 3
Escribe una frase: mi␣ejemplo↵
Palabras: 2
Escribe una frase: ejemplo↵
Palabras: 1
Escribe una frase: otro␣␣ejemplo␣↵
Palabras: 2
Escribe una frase: ↵
```

¡Perfecto! Ya está. ¿Seguro? Mmmm. Los espacios en blanco dieron problemas al final de la cadena. ¿Serán problemáticos también al principio de la cadena? Probemos:

```
Escribe una frase: ␣ejemplo␣↵
Palabras: 2
Escribe una frase: ↵
```

Sí, ¡qué horror! ¿Por qué falla ahora? El problema radica en la inicialización de *anterior* (línea 4). Hemos dado una cadena vacía como valor inicial y eso hace que, si la cadena empieza por un blanco, la condición de la línea 6 se evalúe a *cierto* para el primer carácter, incrementando así la variable *cambios* (línea 7) la primera vez que iteramos el bucle. Podríamos evitarlo modificando la inicialización de la línea 4: un espacio en blanco nos vendría mejor como valor inicial de *anterior*.

palabras.py
```
1 cadena = input('Escribe␣una␣frase:␣')
2 while cadena != '':
3     cambios = 0
4     anterior = '␣'
5     for carácter in cadena:
6         if carácter == '␣' and anterior != '␣':
7             cambios += 1
8         anterior = carácter
9
10    if cadena[-1] == '␣':
11        cambios -= 1
12
13    palabras = cambios + 1
14    print('Palabras:', palabras)
15
16    cadena = input('Escribe␣una␣frase:␣')
```

Ahora sí:

```
Escribe una frase: una␣dos␣tres↵
Palabras: 3
Escribe una frase: mi␣ejemplo↵
Palabras: 2
Escribe una frase: ejemplo↵
Palabras: 1
Escribe una frase: otro␣␣ejemplo␣↵
Palabras: 2
Escribe una frase: ␣ejemplo␣↵
Palabras: 1
Escribe una frase: ↵
```

▶ 155 ¿Funciona el programa cuando introducimos una cadena formada solo por espacios en blanco? ¿Por qué? Si su comportamiento no te parece normal, corrígelo.

El ejemplo que hemos desarrollado tiene un doble objetivo didáctico. Por una parte, familiarizarte con las cadenas; por otra, que veas cómo se resuelve un problema poco a poco. Primero hemos analizado el problema en busca de una solución sencilla (contar espacios en blanco). Después hemos implementado nuestra primera solución y la hemos probado con varios ejemplos. Los ejemplos que nos hemos puesto no son solo los más sencillos, sino aquellos que pueden hacer «cascar» el programa (en nuestro caso, poner dos o más espacios en blanco seguidos). Detectar ese error nos ha conducido a una «mejora» del programa (en realidad, una corrección): no debíamos contar espacios en blanco, sino transiciones de «no blanco» a espacio en blanco. Nuevamente hemos puesto a prueba el programa y hemos encontrado casos para los que falla (espacios al final de la cadena). Un nuevo refinamiento ha permitido tratar el fallo y, otra vez, hemos encontrado un caso no contemplado (espacios al principio de la cadena) que nos ha llevado a un último cambio del programa. Fíjate en que cada vez que hemos hecho un cambio al programa hemos vuelto a introducir todos los casos que ya habíamos probado (al modificar un programa es posible que deje de funcionar para casos en los que ya iba bien) y hemos añadido uno nuevo que hemos sospechado que podía ser problemático. Así es como se llega a la solución final: siguiendo un proceso reiterado de análisis, prueba y error. Durante ese proceso el programador debe «jugar» en dos «equipos» distintos:

- a ratos juega en el equipo de los programadores y trata de encontrar la mejor solución al problema propuesto;

- y a ratos juega en el equipo de los usuarios y pone todo su empeño en buscar configuraciones especiales de los datos de entrada que provoquen fallos en el programa.

▶ 156 Modifica el programa para que base el cómputo de palabras en el número de transiciones de blanco a no blanco en lugar de en el número de transiciones de no blanco a blanco. Comprueba si tu programa funciona en toda circunstancia.

▶ 157 Nuestro aprendiz aventajado propone esta otra solución al problema de contar palabras:

palabras.py

```
cadena = input('Escribe␣una␣frase:␣')
while cadena != '':
    cambios = 0
    for i in range(1, len(cadena)):
        if cadena[i] == '␣' and cadena[i-1] != '␣':
            cambios = cambios + 1

    if cadena[-1] == '␣':
        cambios -= 1

    palabras = cambios + 1
    print('Palabras:', palabras)

    cadena = input('Escribe␣una␣frase:␣')
```

¿Es correcta?

▶ 158 Diseña un programa que lea una cadena y un número entero k y nos diga cuántas palabras tienen una longitud de k caracteres.

▶ 159 Diseña un programa que lea una cadena y un número entero k y nos diga si *alguna* de sus palabras tiene una longitud de k caracteres.

▶ 160 Diseña un programa que lea una cadena y un número entero k y nos diga si *todas* sus palabras tienen una longitud de k caracteres.

▶ 161 Escribe un programa que lea una cadena y un número entero k y muestre el mensaje «`Hay palabras largas`» si alguna de las palabras de la cadena es de longitud mayor o igual que k, y «`No hay palabras largas`» en caso contrario.

▶ 162 Escribe un programa que lea una cadena y un número entero k y muestre el mensaje «`Todas son cortas`» si todas las palabras de la cadena son de longitud estrictamente menor que k, y «`Hay alguna palabra larga`» en caso contrario.

▶ 163 Escribe un programa que lea una cadena y un número entero k y muestre el mensaje «`Todas las palabras son largas`» si todas las palabras de la cadena son de longitud mayor o igual que k, y «`Hay alguna palabra corta`» en caso contrario.

▶ 164 Diseña un programa que muestre la cantidad de dígitos que aparecen en una cadena introducida por teclado. La cadena `'un␣1␣y␣un␣20'`, por ejemplo, tiene 3 dígitos: un 1, un 2 y un 0.

▶ 165 Diseña un programa que muestre la cantidad de números que aparecen en una cadena leída de teclado. ¡Ojo! Con número no queremos decir dígito, sino número propiamente dicho, es decir, secuencia de dígitos. La cadena `'un␣1,␣un␣201␣y␣2␣unos'`, por ejemplo, tiene 3 números: el 1, el 201 y el 2.

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

▶ 166 Diseña un programa que indique si una cadena leída de teclado está bien formada como número entero. El programa escribirá «`Es entero`» en caso afirmativo y «`No es entero`» en caso contrario.

Por ejemplo, para `'12'` mostrará «`Es entero`», pero para `'1␣2'` o `'a'` mostrará «`No es entero`».

▶ 167 Diseña un programa que indique si una cadena introducida por el usuario está bien formada como identificador de variable. Si lo está, mostrará el texto «`Identificador válido`» y si no, «`Identificador inválido`».

▶ 168 Diseña un programa que indique si una cadena leída por teclado está bien formada como número flotante.

Prueba el programa con estas cadenas: `'3.1'`, `'3.'`, `'.1'`, `'1e+5'`, `'-10.2E3'`, `'3.1e-2'`, `'.1e01'`. En todos los casos deberá indicar que se trata de números flotantes correctamente formados.

▶ 169 Un texto está bien parentizado si por cada paréntesis abierto hay otro más adelante que lo cierra. Por ejemplo, la cadena

`'Esto␣(es␣(un)␣(ejemplo␣(de)␣((cadena)␣bien))␣parentizada).'`

está bien parentizada, pero no lo están estas otras:

`'una␣cadena)'` `'(una␣cadena'` `'(una␣(cadena)'` `')una(␣cadena'`

Diseña un programa que lea una cadena y nos diga si la cadena está bien o mal parentizada.

▶ 170 Implementa un programa que lea de teclado una cadena que represente un número binario. Si algún carácter de la cadena es distinto de `'0'` o `'1'`, el programa advertirá al usuario de que la cadena introducida no representa un número binario y pedirá de nuevo la lectura de la cadena.

5.1.7. Otro ejemplo: un programa de conversión de binario a decimal

Nos proponemos diseñar un programa que reciba una cadena compuesta por ceros y unos y muestre un número: el que corresponde al valor decimal de la cadena si interpretamos esta como un número codificado en binario. Por ejemplo, nuestro programa mostrará el valor 13 para la cadena `'1101'`.

Empezaremos por plantearnos cómo haríamos manualmente el cálculo. Podemos recorrer la cadena de izquierda a derecha e ir considerando el aporte de cada bit al número global. El n-ésimo bit contribuye al resultado con el valor 2^{n-1} si vale `'1'`, y con el valor 0 si vale `'0'`. Pero, ¡ojo!, cuando decimos n-ésimo bit, no nos referimos al n-ésimo carácter de la cadena. Por ejemplo, la cadena `'100'` tiene su *tercer* bit a 1, pero ese es el carácter que ocupa la *primera* posición de la cadena (la que tiene índice 0), no la tercera. Podemos recorrer la cadena de izquierda a derecha e ir llevando la cuenta del número de bit actual en una variable:

```
decimal.py
bits = input('Dame␣un␣número␣binario:␣')

n = len(bits)
valor = 0
for bit in bits:
    if bit == '1':
        valor = valor + 2 ** (n-1)
    n -= 1

print('Su␣valor␣decimal␣es', valor)
```

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

▶ 171 Haz una traza para las cadenas '1101' y '010'.

▶ 172 Una vez más, nuestro aprendiz ha diseñado un programa diferente:

```
decimal.py
bits = input('Dame␣un␣número␣binario:␣')

valor = 0
for bit in bits:
    if bit == '1':
        valor = 2 * valor + 1
    else:
        valor = 2 * valor

print('Su␣valor␣decimal␣es', valor)
```

¿Es correcto? Haz trazas para las cadenas '1101' y '010'.

▶ 173 ¿Y esta otra versión? ¿Es correcta?

```
decimal.py
bits = input('Dame␣un␣número␣binario:␣')

valor = 0
for bit in bits:
    if bit == '1':
        valor += valor + 1
    else:
        valor += valor

print('Su␣valor␣decimal␣es', valor)
```

Haz trazas para las cadenas '1101' y '010'.

▶ 174 ¿Y esta otra? ¿Es correcta?

```
decimal.py
bits = input('Dame␣un␣número␣binario:␣')

valor = 0
for bit in bits:
    valor += valor + int(bit)

print('Su␣valor␣decimal␣es', valor)
```

Haz trazas para las cadenas '1101' y '010'.

▶ 175 ¿Qué pasa si introducimos una cadena con caracteres que no pertenecen al conjunto de dígitos binarios como, por ejemplo, '101a2'? Modifica el programa para que, en tal caso, muestre en pantalla el mensaje «Número binario mal formado» y solicite nuevamente la introducción de la cadena.

▶ 176 Diseña un programa que convierta una cadena de dígitos entre el «0» y el «7» al valor correspondiente a una interpretación de dicha cadena como número en base octal.

▶ 177 Diseña un programa que convierta una cadena de dígitos o letras entre la «a» y la «f» al valor correspondiente a una interpretación de dicha cadena como número en base hexadecimal.

▶ 178 Diseña un programa que reciba una cadena que codifica un número en octal, decimal o hexadecimal y muestre el valor de dicho número. Si la cadena empieza por «0x» o «0X» se interpretará como un número hexadecimal (ejemplo: '0xff' es 255); si no, si empieza por «0o» o «0O», la cadena se interpretará como un número octal (ejemplo: '0o17' es 15); y si no, se interpretará como un número decimal (ejemplo: '99' es 99).

▶ 179 Diseña un programa que lea un número entero y muestre una cadena con su representación octal.

▶ 180 Diseña un programa que lea una cadena que representa un número codificado en base 8 y muestre por pantalla su representación en base 2.

5.1.8. A vueltas con las cadenas: inversión de una cadena

Recuerda del capítulo 2 que el operador + puede trabajar con cadenas y denota la operación de concatenación, que permite obtener la cadena que resulta de unir otras dos:

```
>>> 'abc' + 'def'
'abcdef'
```

Vamos a utilizar este operador en el siguiente ejemplo: un programa que lee una cadena y muestra su inversión en pantalla. El programa se ayudará de una cadena auxiliar, inicialmente vacía, en la que iremos introduciendo los caracteres de la cadena original, pero de atrás hacia adelante.

inversion.py
```python
cadena = input('Introduce una cadena: ')

inversión = ''
for carácter in cadena:
    inversión = carácter + inversión

print('Su inversión es:', inversión)
```

Probemos el programa:

```
Introduce una cadena: uno
Su inversión es: onu
```

▶ 181 Una palabra es «alfabética» si todas sus letras están ordenadas alfabéticamente. Por ejemplo, «amor», «chino» e «himno» son palabras «alfabéticas». Diseña un programa que lea una palabra y nos diga si es alfabética o no.

▶ 182 Diseña un programa que nos diga si una cadena es palíndromo o no. Una cadena es palíndromo si se lee igual de izquierda a derecha que de derecha a izquierda. Por ejemplo, 'ana' es un palíndromo.

▶ 183 Una frase es palíndromo si se lee igual de izquierda a derecha que de derecha a izquierda, pero obviando los espacios en blanco y los signos de puntuación. Por ejemplo, las cadenas 'sé verla al revés', 'anita lava la tina', 'luz azul' y 'la ruta natural' contienen frases palíndromas. Diseña un programa que diga si una frase es o no es palíndroma.

▶ 184 Probablemente el programa que has diseñado para el ejercicio anterior falle ante frases palíndromas como estas: «Dábale arroz a la zorra el abad», «Salta Lenín el atlas», «Amigo, no gima», «Átale, demoníaco Caín, o me delata», «Anás usó tu auto, Susana», «A Mercedes, ese de crema», «A mamá Roma le aviva el amor a papá, y a papá Roma le aviva el amor a mamá» y «¡arriba la birra!», pues hemos de comparar ciertas letras con sus versiones acentuadas, o mayúsculas o la apertura de exclamación con su cierre. Modifica tu programa para que identifique correctamente frases palíndromas en las que pueden aparecer letras mayúsculas, vocales acentuadas y la vocal «u» con diéresis.

▶ 185 Hay un tipo de pasatiempos que propone descifrar un texto del que se han suprimido las vocales. Por ejemplo, el texto «`.n .j.mpl. d. p.s.t..mp.s`», se descifra sustituyendo cada punto con una vocal del texto. La solución es «`un ejemplo de pasatiempos`». Diseña un programa que ayude al creador de pasatiempos. El programa recibirá una cadena y mostrará otra en la que cada vocal ha sido reemplazada por un punto.

▶ 186 El nombre de un fichero es una cadena que puede tener lo que denominamos una extensión. La extensión de un nombre de fichero es la serie de caracteres que suceden al último punto presente en la cadena. Si el nombre no tiene ningún punto, asumiremos que su extensión es la cadena vacía. Haz un programa que solicite el nombre de un fichero y muestre por pantalla los caracteres que forman su extensión. Prueba la validez de tu programa pidiendo que muestre la extensión de los nombres de fichero `documento.doc` y `tema.1.tex`, que son `doc` y `tex`, respectivamente.

▶ 187 Haz un programa que lea dos cadenas que representen sendos números binarios. A continuación, el programa mostrará el número binario que resulta de sumar ambos (y que será otra cadena). Si, por ejemplo, el usuario introduce las cadenas `'100'` y `'111'`, el programa mostrará como resultado la cadena `'1011'`.

(Nota: El procedimiento de suma con acarreo que implementes deberá trabajar directamente con la representación binaria leída).

▶ 188 Una de las técnicas de criptografía más rudimentarias consiste en sustituir cada uno de los caracteres por otro situado n posiciones más a la derecha en el abecedario. Si $n = 2$, por ejemplo, sustituiremos la «a» por la «c», la «b» por la «d», y así sucesivamente. El problema que aparece en las últimas n letras del alfabeto tiene fácil solución: en el ejemplo, la letra «y» se sustituirá por la «a» y la letra «z» por la «b». La sustitución debe aplicarse a las letras minúsculas y mayúsculas y a los dígitos (el «0» se sustituye por el «2», el «1» por el «3» y así hasta llegar al «8», que se sustituye por el «0», y el «9», que se sustituye por el «1»).

Diseña un programa que lea un texto y el valor de n y muestre su versión criptografiada.

▶ 189 Diseña un programa que lea un texto criptografiado siguiendo la técnica descrita en el apartado anterior y el valor de n utilizado al encriptar para mostrar ahora el texto decodificado.

5.1.9. Subcadenas: el operador de corte

Desarrollemos un último ejemplo: un programa que, dados una cadena y dos índices i y j, muestra la (sub)cadena formada por todos los caracteres entre el que tiene índice i y el que tiene índice j, incluyendo al primero pero no al segundo.

La idea básica consiste en construir una nueva cadena que, inicialmente, está vacía. Con un recorrido por los caracteres comprendidos entre los de índices i y $j-1$ iremos añadiendo caracteres a la cadena. Vamos con una primera versión:

subcadena.py

```
cadena = input('Dame una cadena: ')
i = int(input('Dame un número: '))
j = int(input('Dame otro número: '))

subcadena = ''
for k in range(i, j):
    subcadena += cadena[k]

print('La subcadena entre {0} y {1} es {2}.'.format(i, j, subcadena))
```

Usémosla:

```
Dame una cadena: Ejemplo↵
Dame un número: 2↵
Dame otro número: 5↵
```

```
La subcadena entre 2 y 5 es emp.
```

¿Falla algo en nuestro programa? Sí: es fácil cometer un error de indexación. Por ejemplo, al ejecutar el programa con la cadena `'Ejemplo'` y los índices 3 y 20 se cometerá un error, pues 20 es mayor que la longitud de la cadena. Corrijamos ese problema:

subcadena.py

```python
cadena = input('Dame una cadena: ')
i = int(input('Dame un número: '))
j = int(input('Dame otro número: '))

if j > len(cadena):
    final = len(cadena)
else:
    final = j

subcadena = ''
for k in range(i, final):
    subcadena += cadena[k]

print('La subcadena entre {0} y {1} es {2}.'.format(i, j, subcadena))
```

▶ 190 ¿Y si se introduce un valor de *i* negativo? Corrige el programa para que detecte esa posibilidad e interprete un índice inicial negativo como el índice 0.

▶ 191 ¿No será también problemático que introduzcamos un valor del índice *i* mayor o igual que el de *j*? ¿Se producirá entonces un error de ejecución? ¿Por qué?

▶ 192 Diseña un programa que, dados una cadena *c*, un índice *i* y un número *n*, muestre la subcadena de *c* formada por los *n* caracteres que empiezan en la posición de índice *i*.

Hemos visto cómo construir una subcadena carácter a carácter. Esta es una operación frecuente en los programas que manejan información textual, así que Python ofrece un operador predefinido que facilita esa labor: el *operador de corte* (en inglés, «slicing operator»). La notación es un tanto peculiar, pero cómoda una vez te acostumbras a ella. Fíjate en este ejemplo:

```
>>> a = 'Ejemplo'
>>> a[2:5]
'emp'
```

El operador de corte se denota con dos puntos (`:`) que separan dos índices *dentro* de los corchetes del operador de indexación. La expresión *a*[*i*:*j*] significa que se desea obtener la subcadena formada por los caracteres *a*[*i*], *a*[*i*+1], . . . , *a*[*j*−1], (observa que, como en *range*, el valor del último índice se omite).

Ya que se omite el último índice del corte, puede que te resulte de ayuda imaginar que los índices de los elementos se disponen en las fronteras entre elementos consecutivos, como se puede ver en esta figura:

0	1	2	3	4	5	6	7
E	j	e	m	p	l	o	
-7	-6	-5	-4	-3	-2	-1	

Ahí queda claro que *a*[2:5], *a*[-5:5], *a*[2,:-2] y *a*[-5:-2], siendo *a* la cadena de la figura, es la cadena `'emp'`.

Cada índice de corte tiene un valor por defecto, así que puedes omitirlo si te conviene. El corte *a*[:*j*] es equivalente a *a*[0:*j*] y el corte *a*[*i*:] equivale a *a*[*i*:*len*(*a*)].

▶ 193 Si *a* vale `'Ejemplo'`, ¿qué es el corte *a*[:]?

▶ 194 ¿Qué corte utilizarías para obtener los *n* caracteres de una cadena a partir de la posición de índice *i*?

▶ 195 Diseña un programa que, dada una cadena, muestre por pantalla todos sus prefijos. Por ejemplo, dada la cadena `'UJI'`, por pantalla debe aparecer:

```
U
UJ
UJI
```

▶ 196 Diseña un programa que lea una cadena y muestre por pantalla todas sus subcadenas de longitud 3.

▶ 197 Diseña un programa que lea una cadena y un entero *k* y muestre por pantalla todas sus subcadenas de longitud *k*.

▶ 198 Diseña un programa que lea dos cadenas *a* y *b* y nos diga si *b* es un prefijo de *a* o no.

(Ejemplo: `'sub'` es un prefijo de `'subcadena'`).

▶ 199 Diseña un programa que lea dos cadenas *a* y *b* y nos diga si *b* es una subcadena de *a* o no.

(Ejemplo: `'de'` es una subcadena de `'subcadena'`).

▶ 200 Diseña un programa que lea dos cadenas y devuelva el prefijo común más largo de ambas.

(Ejemplo: las cadenas `'politécnico'` y `'polinización'` tienen como prefijo común más largo a la cadena `'poli'`).

▶ 201 Diseña un programa que lea *tres* cadenas y muestre el prefijo común más largo de todas ellas.

(Ejemplo: las cadenas `'politécnico'`, `'polinización'` y `'poros'` tienen como prefijo común más largo a la cadena `'po'`).

Cortes avanzados

Desde la versión 2.3, Python entiende una forma extendida de los cortes. Esta forma acepta tres valores separados por el carácter «:». El tercer valor equivale al tercer parámetro de la función *range*: indica el incremento del índice en cada iteración. Por ejemplo, si *c* contiene la cadena `'Ejemplo'`, el corte *c*`[0:`*len*`(`*c*`):2]` selecciona los caracteres de índice par, o sea, devuelve la cadena `'Eepo'`. El tercer valor puede ser negativo. Ello permite invertir una cadena con una expresión muy sencilla: *c*`[::-1]`. Haz la prueba.

5.1.10. Una aplicación: correo electrónico personalizado

Vamos a desarrollar un programa «útil»: uno que envía textos personalizados por correo electrónico. Deseamos enviar una carta tipo a varios clientes, pero adaptando algunos datos de la misma a los propios de cada cliente. Aquí tienes un ejemplo de carta tipo:

```
Estimado =S =A:
Por la presente le informamos de que nos debe usted la cantidad
de =E euros. Si no abona dicha cantidad antes de 3 días, su nombre
pasará a nuestra lista de morosos.
```

Deseamos sustituir las marcas «=S», «=A» y «=E» por el tratamiento (señor o señora), el apellido y la deuda, respectivamente, de cada cliente y enviarle el mensaje resultante por correo electrónico. Nuestro programa pedirá los datos de un cliente, personalizará el escrito, se lo enviará por correo electrónico y a continuación, si lo deseamos, repetirá el proceso para un nuevo cliente.

Antes de empezar a desarrollar el programa nos detendremos para aprender lo básico del módulo *smtplib*, que proporciona funciones para usar el protocolo de envío de correo electrónico

SMTP (siglas de «Simple Mail Transfer Protocol», o sea, «Protocolo Sencillo de Transferencia de Correo»)[1]. Lo mejor será que estudiemos un ejemplo de uso de la librería y que analicemos lo que hace paso a paso.

ejemplo_smtp.py
```
1  from smtplib import SMTP
2
3  servidor = SMTP('alu-mail.uji.es') # Cambia la cadena por el nombre de tu servidor.
4  remitente = 'al00000@alumail.uji.es'
5  destinatario = 'al99999@alumail.uji.es'
6  mensaje = 'From:␣{0}\nTo:␣{1}\n\n'.format(remitente, destinatario)
7  mensaje += 'Hola.\n'
8  mensaje += 'Hasta␣luego.\n'
9
10 servidor.sendmail(remitente, destinatario, mensaje)
```

Vamos por partes. La primera línea importa la función *SMTP* del módulo *smtplib*. La línea 3 crea una conexión con la máquina servidora (vía la llamada a *SMTP*), que en nuestro ejemplo es **alu-mail@uji.es**, y devuelve un objeto que guardamos en la variable *servidor*. Las líneas 4 y 5 guardan las direcciones de correo del remitente y del destinatario en sendas variables, mientras que las tres líneas siguientes definen el mensaje que vamos a enviar. Así, la línea 6 define las denominadas «cabeceras» («headers») del correo y son obligatorias en el protocolo SMTP (respetando, además, los saltos de línea que puedes apreciar al final de las cadenas). Las dos líneas siguientes constituyen el mensaje en sí mismo. Finalmente, la última línea se encarga de efectuar el envío del correo a través de la conexión almacenada en *servidor* y el método *sendmail*. Eso es todo. Si ejecutamos el programa y tenemos permiso del servidor, **al99999@alumail.uji.es** recibirá un correo de **al00000@alumail.uji.es** con el texto que hemos almacenado en *mensaje*.

Nuestro programa presentará el siguiente aspecto:

spam.py
```
1  from smtplib import SMTP
2
3  servidor = SMTP('alu-mail.uji.es')
4  remitente = 'al00000@alumail.uji.es'
5  texto = 'Estimado␣=S␣=A:\n\n'
6  texto += 'Por␣la␣presente␣le␣informamos␣de␣que␣nos␣debe␣usted␣la␣'
7  texto += 'cantidad␣de␣=E␣euros.␣Si␣no␣abona␣dicha␣cantidad␣antes␣'
8  texto += 'de␣3␣días,␣su␣nombre␣pasará␣a␣nuestra␣lista␣de␣morosos.'
9
10 seguir = 's'
11 while seguir == 's':
12     destinatario = input('Dirección␣del␣destinatario:␣')
13     tratamiento = input('Tratamiento:␣')
14     apellido = input('Apellido:␣')
15     euros = input('Deuda␣(en␣euros):␣')
16
17     mensaje = 'From:␣{0}\nTo:␣{1}\n\n'.format(remitente, destinatario)
18     mensaje += texto personalizado
19
20     servidor.sendmail(remitente, destinatario, mensaje)
21     seguir = input('Si␣desea␣enviar␣otro␣correo,␣pulse␣"s":␣')
```

En la línea 18 hemos dejado un fragmento de programa por escribir: el que se encarga de personalizar el contenido de *texto* con los datos que ha introducido el usuario. ¿Cómo personalizamos el texto? Deberíamos ir copiando los caracteres de *texto* uno a uno en una variable

[1]No pierdas de vista que el objetivo de esta sección es aprender el manejo de cadenas. No te despistes tratando de profundizar ahora en los conceptos del SMTP y las peculiaridades del correspondiente módulo.

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

auxiliar (inicialmente vacía) hasta ver el carácter «=», momento en el que deberemos estudiar el siguiente carácter y, en función de cuál sea, añadir el contenido de *tratamiento*, *apellido* o *euros*.

spam.py
```
from smtplib import SMTP

servidor = SMTP('alu-mail.uji.es')
remitente = 'al00000@alumail.uji.es'
texto = 'Estimado␣=S␣=A:\n\n'
texto += 'Por␣la␣presente␣le␣informamos␣de␣que␣nos␣debe␣usted␣la␣'
texto += 'cantidad␣de␣=E␣euros.␣Si␣no␣abona␣dicha␣cantidad␣antes␣'
texto += 'de␣3␣días,␣su␣nombre␣pasará␣a␣nuestra␣lista␣de␣morosos.'

seguir = 's'
while seguir == 's':
    destinatario = input('Dirección␣del␣destinatario:␣')
    tratamiento = input('Tratamiento:␣')
    apellido = input('Apellido:␣')
    euros = input('Deuda␣(en␣euros):␣')

    mensaje = 'From:␣{0}\nTo:␣{1}\n\n'.format(remitente, destinatario)

    personalizado = ''
    i = 0
    while i < len(texto):
        if texto[i] != '=':
            personalizado += texto[i]
        else:
            if texto[i+1] == 'A':
                personalizado += apellido
                i = i + 1
            elif texto[i+1] == 'E':
                personalizado += euros
                i = i + 1
            elif texto[i+1] == 'S':
                personalizado += tratamiento
                i = i + 1
            else:
                personalizado += '='
        i = i + 1
    mensaje += personalizado

    servidor.sendmail(remitente, destinatario, mensaje)
    seguir = input('Si␣desea␣enviar␣otro␣correo,␣pulse␣\'s\':␣')
```

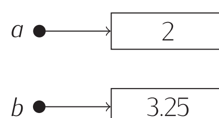
▶ 202 El programa no funcionará bien con cualquier carta. Por ejemplo, si la variable *texto* vale `'Hola␣=A.␣='` el programa falla. ¿Por qué? ¿Sabrías corregir el programa?

5.1.11. Referencias a cadenas

En el apartado 2.4 hemos representado las variables y su contenido con diagramas de cajas. Por ejemplo, las siguientes asignaciones:

```
>>> a = 2↵
>>> b = 3.25↵
```

conducen a una disposición de la información en la memoria que mostramos gráficamente así:



Buscando texto en cadenas

Estudiamos los aspectos fundamentales de las cadenas y montamos «a mano» las operaciones más sofisticadas. Por ejemplo, hemos estudiado la indexación y la utilizamos, en combinación con un bucle, para buscar un carácter determinado en una cadena. Pero esa es una operación muy frecuente, así que Python la trae «de serie».

El método *find* recibe una cadena y nos dice si esta aparece o no en la cadena sobre la que se invoca. Si está, nos devuelve el índice de su primera aparición. Si no está, devuelve el valor −1. Atención a estos ejemplos:

```
>>> c = 'Un␣ejemplo␣=A.'↵
>>> c.find('=')↵
11
>>> c.find('ejem')↵
3
>>> c.find('z')↵
-1
```

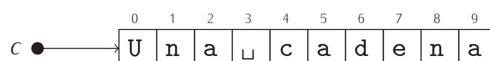
Útil, ¿no? Pues hay muchos más métodos que permiten realizar operaciones complejas con enorme facilidad. Encontrarás, entre otros, métodos para sustituir un fragmento de texto por otro, para saber si todos los caracteres son minúsculas (o mayúsculas), para saber si empieza o acaba con un texto determinado, etc. Cuantos más métodos avanzados conozcas, más productivo serás. ¿Que dónde encontrarás la relación de métodos? En la documentación de Python. Acostúmbrate a manejarla.

Decimos que *a apunta* al valor 2 y que *b apunta* al valor 3.25. La flecha recibe el nombre de puntero o referencia.

Con las cadenas representaremos los valores desglosando cada uno de sus caracteres en una caja individual con un índice asociado. El resultado de una asignación como esta:

```
>>> c = 'Una␣cadena'↵
```

se representará del siguiente modo:

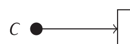


Decimos que la variable *c apunta* a la cadena `'Una␣cadena'`, que es una secuencia de caracteres.

La cadena vacía no ocupa ninguna celda de memoria y la representamos gráficamente de un modo especial. Una asignación como esta:

```
>>> c = ''↵
```

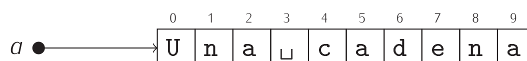
se representará del siguiente modo:



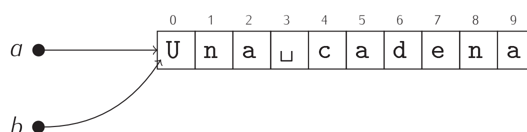
Que las variables contengan referencias a los datos y no los propios datos es muy útil para aprovechar la memoria del ordenador. El siguiente ejemplo te ilustrará el ahorro que se consigue.

```
>>> a = 'Una␣cadena'↵
>>> b = a↵
```

Tras ejecutar la primera acción tenemos:

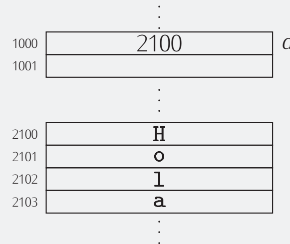


Y después de ejecutar la segunda:

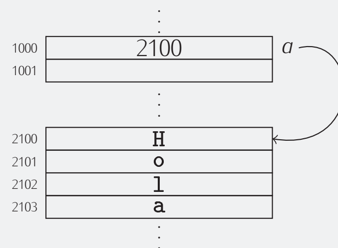


Las referencias son direcciones de memoria (I)

Vamos a darte una interpretación de las referencias que, aunque constituye una simplificación de la realidad, te permitirá entender qué son. Ya dijimos en el capítulo 1 que la memoria del computador se compone de una serie de celdas numeradas con sus direcciones. En cada celda cabe un escalar. La cadena `'Hola'` ocupa cuatro celdas, una por cada carácter. Por otra parte, una variable solo puede contener un escalar. Como la dirección de memoria es un número y, por tanto, un escalar, el «truco» consiste en almacenar en la variable la dirección de memoria en la que empieza la cadena. Fíjate en este ejemplo en el que una variable ocupa la dirección de memoria 1000 y «contiene» la cadena `'Hola'`:



Como puedes ver, en realidad la cadena ocupa posiciones consecutivas a partir de una dirección determinada (en el ejemplo, la 2100) y la variable contiene el valor de dicha referencia. La flecha de los diagramas hace más «legibles» las referencias:

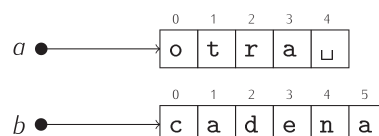


¡Tanto *a* como *b apuntan* a la misma cadena! Al asignar a una variable la cadena contenida en otra *únicamente se copia su referencia* y no cada uno de los caracteres que la componen. Si se hiciera del segundo modo, la memoria ocupada y el tiempo necesarios para la asignación serían tanto mayores cuanto más larga fuera la cadena. El método escogido únicamente copia el valor de la referencia, así que es independiente de la longitud de la cadena (y prácticamente instantáneo).

Has de tener en cuenta, pues, que una asignación únicamente altera el valor de un puntero. Pero otras operaciones con cadenas comportan la reserva de nueva memoria. Tomemos por caso el operador de concatenación. La concatenación toma dos cadenas y forma *una cadena nueva* que resulta de unir ambas, es decir, reserva memoria para una nueva cadena. Veamos paso a paso cómo funciona el proceso con un par de ejemplos. Fíjate en estas sentencias:

```
>>> a = 'otra␣'↵
>>> b = 'cadena'↵
>>> c = a + b↵
```

Podemos representar gráficamente el resultado de la ejecución de las dos primeras sentencias así:

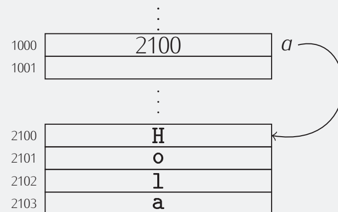


Las referencias son direcciones de memoria (y II)

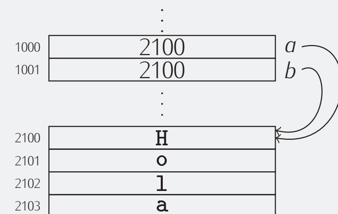
Veamos qué ocurre cuando dos variables comparten referencia. El ejemplo que hemos desarrollado en el texto estudia el efecto de estas dos asignaciones:

```
>>> a = 'Una␣cadena'↵
>>> b = a↵
```

Como vimos antes, la primera asignación conduce a esta situación:

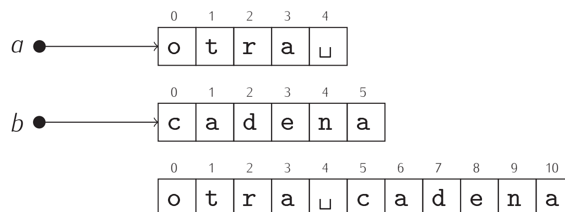


Pues bien, la segunda asignación copia en la dirección de *b* (que suponemos es la 1001) el valor que hay almacenado en la dirección de *a*, es decir, el valor 2100:

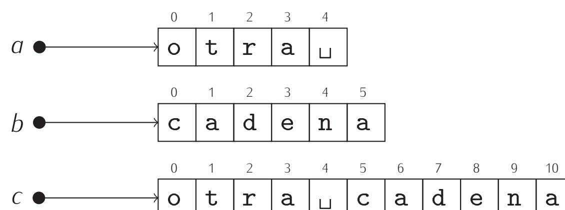


Copiar un valor escalar de una posición de memoria a otra es una acción muy rápida.

Analicemos ahora la tercera sentencia. Primero, Python evalúa la expresión *a* + *b*, así que reserva un bloque de memoria con espacio para 11 caracteres y copia en ellos los caracteres de *a* seguidos de los caracteres de *b*:



Y ahora que ha creado la nueva cadena, se ejecuta la asignación en sí, es decir, se hace que *c* apunte a la nueva cadena:

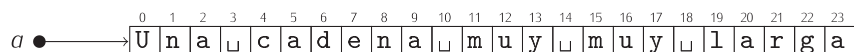


El orden en el que ocurren las cosas tiene importancia para entender cómo puede verse afectada la velocidad de ejecución de un programa por ciertas operaciones. Tomemos por caso estas dos órdenes:

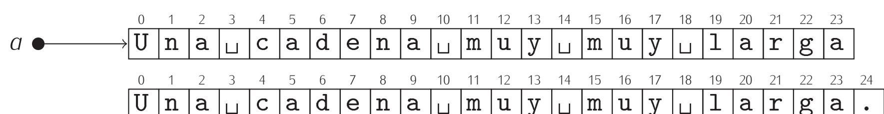
```
>>> a = 'una␣cadena␣muy␣muy␣larga'↵
```

```
>>> a = a + '.'↵
```

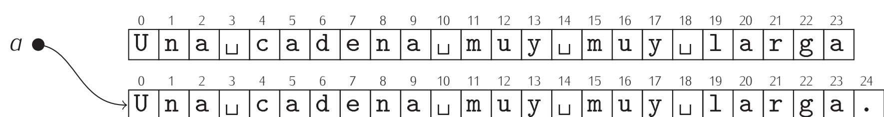
A simple vista parece que la primera sentencia será más lenta en ejecución que la segunda, pues comporta la reserva de una zona de memoria que puede ser grande (imagina si la cadena tuviera mil o incluso cien mil caracteres), mientras que la segunda sentencia se limita a añadir un solo carácter. Pero no es así: ambas tardan casi lo mismo. Veamos cuál es la razón. La primera sentencia reserva memoria para 24 caracteres, los guarda en ella y hace que *a* apunte a dicha zona:



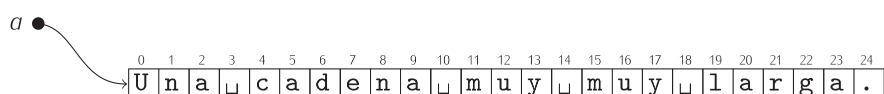
Y ahora veamos paso a paso qué ocurre al ejecutar la segunda sentencia. En primer lugar, se evalúa la parte derecha, es decir, se reserva espacio para 25 caracteres y se copian en él los 24 caracteres de *a* y el carácter punto:



Y ahora, al ejecutar la asignación, la variable *a* deja de apuntar a la zona de memoria original para apuntar a la nueva zona de memoria:



Como la zona inicial de memoria ya no se usa para nada, Python la «libera», es decir, considera que está disponible para futuras operaciones, con lo que, a efectos prácticos, desaparece:

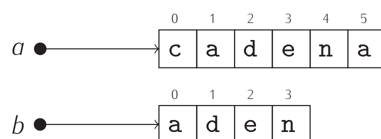


Como puedes ver, la sentencia que consiste en añadir un simple punto a una cadena es más costosa en tiempo que la que comporta una asignación a una variable de esa misma cadena.

El operador con asignación += actúa exactamente igual con cadenas, así que sustituir la última sentencia por *a* += '.' presenta el mismo problema.

El operador de corte también reserva una nueva zona de memoria:

```
>>> a = 'cadena'↵
>>> b = a[1:-1]↵
```



▶ 203 Dibuja un diagrama con el estado de la memoria tras ejecutar estas sentencias:

```
>>> a = 'cadena'↵
>>> b = a[2:3]↵
>>> c = b + ''↵
```

▶ 204 Dibuja diagramas que muestren el estado de la memoria paso a paso para esta secuencia de asignaciones.

```
>>> a = 'ab'↵
```

```
>>> a *= 3
>>> b = a
>>> c = a[:]
>>> c = c + b
```

¿Qué se mostrará por pantalla si imprimimos *a*, *b* y *c* al final?

5.2. Listas

El concepto de secuencia es muy potente y no se limita a las cadenas. Python nos permite definir secuencias de valores de cualquier tipo. Por ejemplo, podemos definir secuencias de números enteros o flotantes, o incluso de cadenas. Hablamos entonces de *listas*. En una lista podemos, por ejemplo, registrar las notas de los estudiantes de una clase, la evolución de la temperatura hora a hora, los coeficientes de un polinomio, la relación de nombres de personas asistentes a una reunión, etc.

Python sigue una notación especial para representar las listas. Los valores de una lista deben estar encerrados entre corchetes y separados por comas. He aquí una lista con los números del 1 al 3:

```
>>> [1, 2, 3]
[1, 2, 3]
```

Podemos asignar listas a variables:

```
>>> a = [1, 2, 3]
>>> a
[1, 2, 3]
```

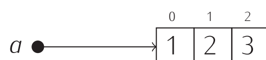
Los elementos que forman una lista también pueden ser cadenas.

```
>>> nombres = ['Juan', 'Antonia', 'Luis', 'María']
```

Y también podemos usar expresiones para calcular el valor de cada elemento de una lista:

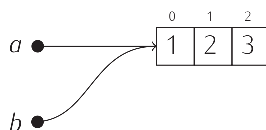
```
>>> a = [1, 1+1, 6//2]
>>> a
[1, 2, 3]
```

Python almacena las listas del mismo modo que las cadenas: mediante referencias (punteros) a la secuencia de elementos. Así, el último ejemplo hace que la memoria presente un aspecto como el que muestra el siguiente diagrama:



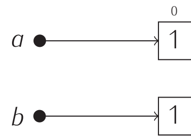
La asignación a una variable del contenido de otra variable que almacena una (referencia a una) lista supone la copia de, únicamente, su referencia, así que ambas acaban apuntando a la misma zona de memoria:

```
>>> a = [1, 2, 3]
>>> b = a
```



La lista que contiene un solo elemento presenta un aspecto curioso:

```
>>> a = [1]
>>> b = 1
```



Observa que no es lo mismo [10] que 10. [10] es la lista cuyo único elemento es el entero 10, y 10 es un entero. Gráficamente hemos destacado la lista disponiendo encima de la celda su índice. Si pedimos a Python que nos muestre el contenido de las variables *a* y *b*, veremos que la representación de la lista que contiene un escalar y la del escalar son diferentes:

```
>>> a = [1]↵
>>> b = 1↵
>>> print(a, b)↵
[1] 1
```

La lista siempre se muestra encerrada entre corchetes.

Del mismo modo que hay una cadena vacía, existe también una *lista vacía*. La lista vacía se denota así: [] y la representamos gráficamente como la cadena vacía:

```
>>> a = []↵
>>> print(a)↵
[]
```



5.2.1. Cosas que, sin darnos cuenta, ya sabemos sobre las listas

Una ventaja de Python es que proporciona operadores y funciones similares para trabajar con tipos de datos similares. Las cadenas y las listas tienen algo en común: ambas son *secuencias* de datos, así pues, muchos de los operadores y funciones que trabajan sobre cadenas también lo hacen sobre listas. Por ejemplo, la función *len*, aplicada sobre una lista, nos dice cuántos elementos la integran:

```
>>> a = [1, 2, 3]↵
>>> len(a)↵
3
>>> len([0, 1, 10, 5])↵
4
>>> len([10])↵
1
```

La longitud de la lista vacía es 0:

```
>>> len([])↵
0
```

El operador + concatena listas:

```
>>> [1, 2] + [3, 4]↵
[1, 2, 3, 4]
>>> a = [1, 2, 3]↵
>>> [10, 20] + a↵
[10, 20, 1, 2, 3]
```

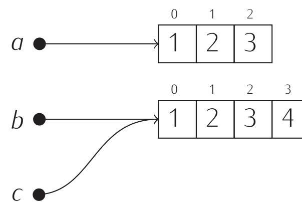
y el operador * repite un número dado de veces una lista:

```
>>> [1, 2] * 3↵
[1, 2, 1, 2, 1, 2]
>>> a = [1, 2, 3]↵
>>> b = [10, 20] + a * 2↵
>>> b ↵
[10, 20, 1, 2, 3, 1, 2, 3]
```

Has de tener en cuenta que tanto + como * generan nuevas listas, sin modificar las originales. Observa este ejemplo:

```
>>> a = [1, 2, 3]↵
>>> b = a + [4]↵
>>> c = b↵
```

La memoria queda así:



¿Ves? La asignación a *b* deja intacta la lista *a* porque apunta al resultado de concatenar algo a *a*. La operación de concatenación no modifica la lista original: reserva memoria para una nueva lista con tantos elementos como resultan de sumar la longitud de las listas concatenadas y, a continuación, copia los elementos de la primera lista seguidos por los de la segunda lista en la nueva zona de memoria. Como asignamos a *b* el resultado de la concatenación, tenemos que *b* apunta a la lista recién creada. La tercera sentencia es una simple asignación a *c*, así que Python se limita a copiar la referencia.

El operador de indexación también es aplicable a las listas:

```
>>> a = [1, 2, 3]↵
>>> a[1]↵
2
>>> a[len(a)-1]↵
3
>>> a[-1]↵
3
```

A veces, el operador de indexación puede dar lugar a expresiones algo confusas a primera vista:

```
>>> [1, 2, 3][0]↵
1
```

En este ejemplo, el primer par de corchetes indica el principio y final de la lista (formada por el 1, el 2 y el 3) y el segundo par indica el índice del elemento al que deseamos acceder (el primero, es decir, el de índice 0).

▶ 205 ¿Qué aparecerá por pantalla al evaluar la expresión `[1][0]`? ¿Y al evaluar la expresión `[][0]`?

De todos modos, no te preocupes por esa notación un tanto confusa: lo normal es que accedas a los elementos de listas que están almacenadas en variables, con lo que rara vez tendrás dudas.

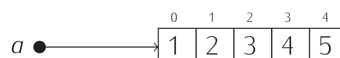
```
>>> a = [1, 2, 3]↵
>>> a[0]↵
1
```

También el operador de corte es aplicable a las listas:

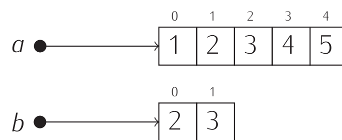
```
>>> a = [1, 2, 3]↵
>>> a[1:-1]↵
[2]
>>> a[1:]↵
[2, 3]
```

Has de tener en cuenta que un corte siempre se extrae copiando un fragmento de la lista, por lo que comporta la reserva de memoria para crear una nueva lista. Analiza la siguiente secuencia de acciones y sus efectos sobre la memoria:

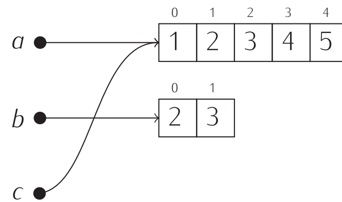
```
>>> a = [1, 2, 3, 4, 5]↵
```



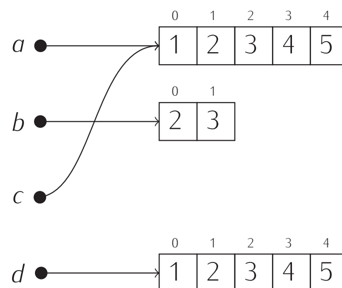

```
>>> a = [1, 2, 3, 4, 5]↵
>>> b = a[1:3]↵
```




```
>>> a = [1, 2, 3, 4, 5]↵
>>> b = a[1:3]↵
>>> c = a↵
```




```
>>> a = [1, 2, 3, 4, 5]↵
>>> b = a[1:3]↵
>>> c = a↵
>>> d = a[:]↵
```



Si deseas asegurarte de que trabajas con una copia de una lista y no con la misma lista (a través de una referencia) utiliza el operador de corte en la asignación.

▶ 206 Hemos asignado a x la lista `[1, 2, 3]` y ahora queremos asignar a y una copia. Podríamos hacer y `=` x`[:]`, pero parece que y `=` x `+ []` también funciona. ¿Es así? ¿Por qué?

El iterador **for-in** también recorre los elementos de una lista:

```
>>> for i in [1, 2, 3]:↵
...     print(i)↵
... ↵
1
2
3
```

De hecho, ya hemos utilizado bucles que recorren secuencias de valores:

```
>>> for i in range(1, 4):↵
...     print(i)↵
... ↵
1
2
3
```

Recuerda que puedes combinar las funciones *list* y *range* para construir cómodamente una lista de valores:

```
>>> a = list(range(1, 4))↵
>>> print(a)↵
[1, 2, 3]
```

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

Una forma corriente de construir listas que contienen réplicas de un mismo valor se ayuda del operador *. Supongamos que necesitamos una lista de 10 elementos, todos los cuales valen 0. Podemos hacerlo así:

```
>>> [0] * 10↵
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

▶ 207 ¿Qué aparecerá por pantalla al ejecutar este programa?

```
print('Principio')
for i in []:
    print('paso', i)
print('y␣fin')
```

▶ 208 ¿Qué aparecerá por pantalla al ejecutar este programa?

```
for i in [1] * 10:
    print(i)
```

5.2.2. Comparación de listas

Los operadores de comparación también trabajan con listas. Parece claro cómo se comportarán operadores como el de igualdad (==) o el de desigualdad (!=):

- si las listas son de talla diferente, resolviendo que las listas son diferentes;
- y si miden lo mismo, comparando elemento a elemento de izquierda a derecha y resolviendo que las dos listas son iguales si todos sus elementos son iguales, y diferentes si hay algún elemento distinto.

Hagamos un par de pruebas con el intérprete de Python:

```
>>> [1, 2, 3] == [1, 2]↵
False
>>> [1, 2, 3] == [1, 2, 3]↵
True
>>> [1, 2, 3] == [1, 2, 4]↵
False
```

Los operadores <, >, <= y >= también funcionan con listas. ¿Cómo? Del mismo modo que con las cadenas, pues al fin y al cabo, tanto cadenas como listas son *secuencias*. Tomemos, por ejemplo, el operador < al comparar las listas [1, 2, 3] y [1, 3, 2], es decir, al evaluar la expresión [1, 2, 3] < [1, 3, 2]. Se empieza por comparar los primeros elementos de ambas listas. Como no es cierto que 1 < 1, pasamos a comparar los respectivos segundos elementos. Como 2 < 3, el resultado es True, sin necesidad de efectuar ninguna comparación adicional.

▶ 209 ¿Sabrías decir que resultados se mostrarán al ejecutar estas sentencias?

```
>>> [1, 2] < [1, 2]↵
>>> [1, 2, 3] < [1, 2]↵
>>> [1, 1] < [1, 2]↵
>>> [1, 3] < [1, 2]↵
>>> [10, 20, 30] > [1, 2, 3]↵
>>> [10, 20, 3] > [1, 2, 3]↵
>>> [10, 2, 3] > [1, 2, 3]↵
>>> [1, 20, 30] > [1, 2, 3]↵
>>> [0, 2, 3] <= [1, 2, 3]↵
>>> [1] < [2, 3]↵
>>> [1] < [1, 2]↵
>>> [1, 2] < [0]↵
```

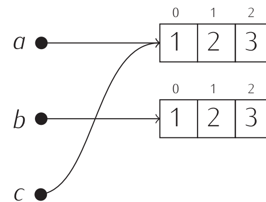
▶ 210 Diseña un programa que tras asignar dos listas a sendas variables nos diga si la primera es menor que la segunda. No puedes utilizar operadores de comparación entre listas para implementar el programa.

5.2.3. El operador is

Hemos visto que las listas conllevan una forma de reservar memoria curiosa: en ocasiones, dos variables apuntan a una misma zona de memoria y en ocasiones no, incluso cuando los datos de ambas variables son idénticos. Fíjate en este ejemplo:

```
>>> a = [1, 2, 3]↵
>>> b = [1, 2, 3]↵
>>> c = a↵
```

Ya hemos visto que, tras efectuar las asignaciones, la memoria quedará así:



¿Qué ocurre si comparamos entre sí los diferentes elementos?

```
>>> a = [1, 2, 3]↵
>>> b = [1, 2, 3]↵
>>> c = a↵
>>> a == b↵
True
>>> a == c↵
True
```

Efectivamente: siempre dice que se trata de listas iguales, y es cierto. Sí, pero, ¿no son «más iguales» las listas *a* y *c* que las listas *a* y *b*? A fin de cuentas, tanto *a* como *c* apuntan exactamente a la misma zona de memoria, mientras que *b* apunta a una zona distinta. Python dispone de un operador de comparación especial que aún no te hemos presentado: is (en español, «es»). El operador is devuelve True si dos objetos son en realidad el mismo objeto, es decir, si residen ambos en la misma zona de memoria, y False en caso contrario.

```
>>> a = [1, 2, 3]↵
>>> b = [1, 2, 3]↵
>>> c = a↵
>>> a is b↵
False
>>> a is c↵
True
```

Python reserva nuevos bloques de memoria conforme evalúa expresiones. Observa este ejemplo:

```
>>> a = [1, 2]↵
>>> a is [1, 2]↵
False
>>> a == [1, 2]↵
True
```

La segunda orden compara la lista almacenada en *a*, que se creó al evaluar una expresión en la orden anterior, con la lista `[1, 2]` que se crea en ese mismo instante, así que is nos dice que ocupan posiciones de memoria diferentes. El operador == sigue devolviendo el valor True, pues aunque sean objetos diferentes son equivalentes elemento a elemento.

▶ 211 ¿Qué ocurrirá al ejecutar estas órdenes Python?

```
>>> a = [1, 2, 3]↵
>>> a is a↵
>>> a + [] is a↵
>>> a + [] == a↵
```

▶ 212 Explica, con la ayuda de un gráfico que represente la memoria, los resultados de evaluar estas expresiones:

```
>>> a = [1, 2, 1]↵
>>> b = [1, 2, 1]↵
>>> (a[0] is b[0]) and (a[1] is b[1]) and (a[2] is b[2])↵
True
>>> a == b↵
True
>>> a is b↵
False
```

▶ 213 ¿Qué ocurrirá al ejecutar estas órdenes Python?

```
>>> [1, 2] == [1, 2]↵
>>> [1, 2] is [1, 2]↵
>>> a = [1, 2, 3]↵
>>> b = [a[0], a[1], a[2]]↵
>>> a == b↵
>>> a is b↵
>>> a[0] == b[1]↵
>>> b is [b[0], b[1], b[2]]↵
```

▶ 214 ¿Qué se muestra por pantalla como respuesta a cada una de estas sentencias Python?

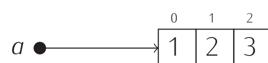
```
>>> a = [1, 2, 3, 4, 5]↵
>>> b = a[1:3]↵
>>> c = a↵
>>> d = a[:]↵
>>> a == c↵
>>> a == d↵
>>> c == d↵
>>> a == b↵
>>> a is c↵
>>> a is d↵
>>> c is d↵
>>> a is b↵
```

5.2.4. Modificación de elementos de listas

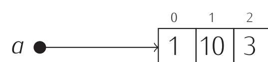
Hasta el momento hemos aprendido a crear listas y a consultar su contenido, bien accediendo a uno cualquiera de sus elementos (mediante indexación), bien recorriendo todos sus elementos (con un bucle **for-in**). En este apartado veremos cómo modificar el contenido de las listas.

Podemos asignar valores a elementos particulares de una lista gracias al operador de indexación:

```
>>> a = [1, 2, 3]↵
```




```
>>> a = [1, 2, 3]↵
>>> a[1] = 10↵
```




```
>>> a = [1, 2, 3]↵
>>> a[1] = 10↵
>>> a↵
[1, 10, 3]
```

Cada celda de una lista es, en cierto modo, una variable autónoma: podemos almacenar en ella un valor y modificarlo a voluntad.

▶ 215 Haz un programa que almacene en una variable *a* la lista obtenida mediante *list*(*range*(1,4)) y, a continuación, la modifique para que cada componente sea igual al cuadrado del componente original. El programa mostrará la lista resultante por pantalla.

▶ 216 Haz un programa que almacene en *a* una lista obtenida con *list*(*range*(1,*n*)), donde *n* es un entero que se pide al usuario, y modifique dicha lista para que cada componente sea igual al cuadrado del componente original. El programa mostrará la lista resultante por pantalla.

▶ 217 Haz un programa que, dada una lista *a* cualquiera, sustituya cualquier elemento negativo por cero.

▶ 218 ¿Qué mostrará por pantalla el siguiente programa?

copias.py
```
a = list(range(0, 5))
b = list(range(0, 5))
c = a
d = b[:]
e = a + b
f = b[:1]
g = b[0]
c[0] = 100
d[0] = 200
e[0] = 300
print(a, b, c, d, e, f, g)
```

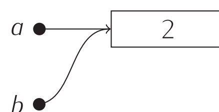
Comprueba con el ordenador la validez de tu respuesta.

5.2.5. Mutabilidad, inmutabilidad y representación de la información en memoria

Python procura no consumir más memoria que la necesaria. Ciertos objetos son inmutables, es decir, no pueden modificar su valor. El número 2 es siempre el número 2. Es un objeto inmutable. Python procura almacenar en memoria una sola vez cada valor inmutable. Si dos o más variables contienen ese valor, sus referencias apuntan a la misma zona de memoria. Considera este ejemplo:

```
>>> a = 1 + 1
>>> b = 2 * 1
```

La memoria presenta, tras esas asignaciones, este aspecto:

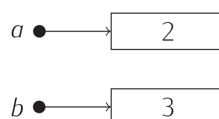


¿Y qué ocurre cuando modificamos el valor de una variable inmutable? No se modifica el contenido de la caja que contiene el valor, sino que el correspondiente puntero pasa a apuntar a una caja con el nuevo valor; y si esta no existe, se crea.

Si a las asignaciones anteriores les sigue una más:

```
>>> a = 1 + 1
>>> b = 2 * 1
>>> b = b + 1
```

la memoria pasa a tener este aspecto:

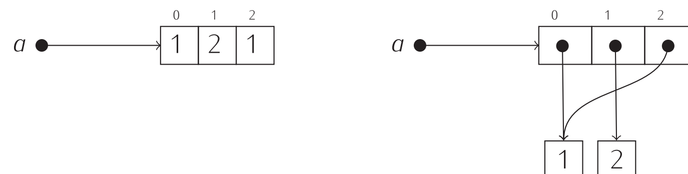


Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

También las cadenas Python son objetos inmutables[2]. Que lo sean tiene efectos sobre las operaciones que podemos efectuar con ellas. La asignación a un elemento de una cadena, por ejemplo, está prohibida, así que Python la señala con un «error de tipo» (*TypeError*):

```
>>> a = 'Hola'↵
>>> a[0] = 'h'↵
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

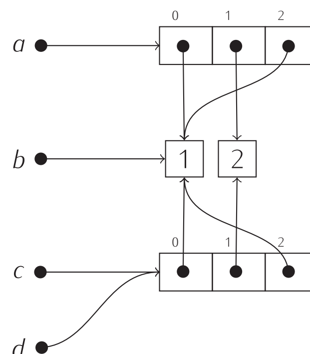
Las listas se comportan de forma diferente: a diferencia de las cadenas, son mutables. De momento te hemos proporcionado una representación de las listas excesivamente simplificada. Hemos representando el resultado de la asignación *a* = [1, 2, 1] como se muestra a la izquierda, cuando lo correcto sería hacerlo como se muestra a la derecha:



La realidad, como ves, es algo complicada: la lista almacena referencias a los valores, y no los propios valores. Pero aún no lo has visto todo. ¿Qué ocurre tras ejecutar estas sentencias?

```
>>> a = [1, 2, 1]↵
>>> b = 1↵
>>> c = [1, 2, 1]↵
>>> d = c↵
```

Nada menos que esto:



Como habrás observado, para cada aparición de un literal de lista, es decir, de una lista expresada explícitamente, (como [1, 2, 1]), Python ha reservado nueva memoria, aunque exista otra lista de idéntico valor. Así pues, *a* = [1, 2, 1] y *c* = [1, 2, 1] han generado sendas reservas de memoria y cada variable apunta a una zona de memoria diferente. Como el contenido de cada celda ha resultado ser un valor inmutable (un entero), se han compartido las referencias a los mismos. El operador **is** nos ayuda a confirmar nuestra hipótesis:

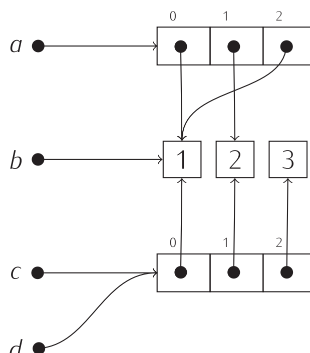
```
>>> a = [1, 2, 1]↵
>>> b = 1↵
>>> c = [1, 2, 1]↵
>>> d = c↵
>>> a[0] is b↵
True
>>> c[-1] is a[0]↵
True
```

Modifiquemos ahora el contenido de una celda de una de las listas:

[2]Aunque los ejemplos que hemos presentado con enteros no son directamente trasladables al caso de las cadenas. Aunque parezca paradójico, Python puede decidir por razones de eficiencia que dos cadenas con idéntico contenido se almacenen por duplicado.

```
>>> a = [1, 2, 1]↵
>>> b = 1↵
>>> c = [1, 2, 1]↵
>>> d = c↵
>>> d[2] = 3↵
```

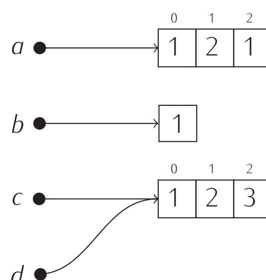
El resultado es este:



▶ 219 Representa el estado de la memoria tras efectuar cada una de las siguientes asignaciones:

```
>>> a = [1, 2, 1]↵
>>> b = 1↵
>>> c = [2, 1, 2]↵
>>> d = c↵
>>> d[2] = 3↵
>>> e = d[:1]↵
>>> f = d[:]↵
>>> f[0] = a[1]↵
>>> f[1] = 1↵
```

Aunque los diagramas que hemos mostrado responden a la realidad, usaremos normalmente su versión simplificada (y, en cierto modo, «falsa»), pues es suficiente para el diseño de la mayor parte de programas que vamos a presentar. Con esta visión simplificada, la última figura se representaría así:



5.2.6. Adición de elementos a una lista

Podemos añadir elementos a una lista, esto es, hacerla crecer. ¿Cómo? Una idea que parece natural, pero que no funciona, es asignar un valor a *a*[*len*(*a*)] (siendo *a* una variable que contiene una lista), pues de algún modo estamos señalando una posición más a la derecha del último elemento. Python nos indicará que estamos cometiendo un error:

```
>>> a = [1, 2, 3]↵
>>> a[len(a)] = 4↵
Traceback (most recent call last):
  File "<input>", line 1, in <module>
IndexError: list assignment index out of range
```

Una idea mejor consiste en utilizar el operador +:

```
>>> a = [1, 2, 3]↵
>>> a = a + 4↵
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list
```

Algo ha ido mal. ¡Claro!, el operador de concatenación trabaja con *dos listas*, no con *una lista y un entero*, así que el elemento a añadir debe formar parte de una lista… aunque esta solo tenga un elemento:

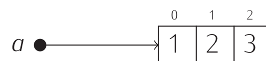
```
>>> a = [1, 2, 3]↵
>>> a = a + [4]↵
>>> a↵
[1, 2, 3, 4]
```

Existe otro modo efectivo de añadir elementos a una lista: mediante el método *append* (que en inglés significa «añadir»). Observa cómo usamos *append*:

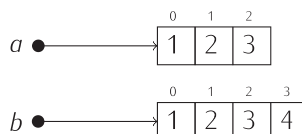
```
>>> a = [1, 2, 3]↵
>>> a.append(4)↵
>>> a↵
[1, 2, 3, 4]
```

Hay una diferencia fundamental entre usar el operador de concatenación + y usar *append*: la concatenación *crea* una nueva lista copiando los elementos de las listas que participan como operandos y *append modifica* la lista original. Observa qué ocurre paso a paso en el siguiente ejemplo:

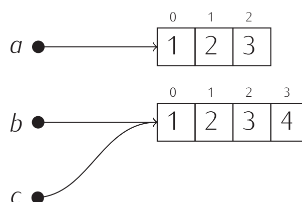
```
>>> a = [1, 2, 3]↵
```



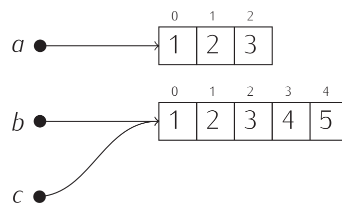

```
>>> a = [1, 2, 3]↵
>>> b = a + [4]↵
```




```
>>> a = [1, 2, 3]↵
>>> b = a + [4]↵
>>> c = b↵
```




```
>>> a = [1, 2, 3]↵
>>> b = a + [4]↵
>>> c = b↵
>>> c.append(5)↵
```




```
>>> a = [1, 2, 3]↵
>>> b = a + [4]↵
>>> c = b↵
>>> c.append(5)↵
>>> print(a)↵
[1, 2, 3]
>>> print(b)↵
[1, 2, 3, 4, 5]
>>> print(c)↵
[1, 2, 3, 4, 5]
```

¿Por qué complicarse la vida con *append*, cuando la concatenación hace lo mismo y nos asegura trabajar con una copia de la memoria? Por eficiencia: es más eficiente hacer *append* que concatenar. Concatenar supone crear una lista nueva en la que se copian todos y cada uno de los elementos de las listas concatenadas. Es decir, la concatenación del siguiente ejemplo supone la copia de 1001 elementos (los 1000 de la lista original y el que añadimos):

```
>>> a = list(range(1000))↵
>>> a = a + [0]↵
```

Sin embargo, el *append* de este otro ejemplo equivalente trabaja sobre la lista original y le añade una celda cuyo contenido es 0:[3]

```
>>> a = list(range(1000))↵
>>> a.append(0)↵
```

En este ejemplo, pues, el *append* ha resultado unas 1000 veces más eficiente que la concatenación.

Desarrollemos un ejemplo práctico. Vamos a escribir un programa que construya una lista con todos los números primos entre 1 y *n*. Como no sabemos a priori cuántos hay, construiremos una lista vacía e iremos añadiendo números primos conforme los vayamos encontrando.

obten_primos.py
```
n = int(input('Introduce␣el␣valor␣máximo:␣'))

primos = []
for número in range(2, n+1):
    # Determinamos si número es primo.
    creo_que_es_primo = True
    for divisor in range(2, número):
        if número % divisor == 0:
            creo_que_es_primo = False
            break
    # Y si es primo, lo añadimos a la lista.
    if creo_que_es_primo:
        primos.append(número)

print(primos)
```

▶ 220 Diseña un programa que construya una lista con los *n* primeros números primos (ojo: no los primos entre 1 y *n*, sino los *n* primeros números primos). ¿Necesitas usar *append*? ¿Puedes reservar en primer lugar una lista con *n* celdas nulas y asignarle a cada una de ellas uno de los números primos?

---

[3]No siempre es *más* eficiente añadir que concatenar. Python puede necesitar memoria para almacenar la lista resultante de añadir un elemento y, entonces, ha de efectuar una copia del contenido de la lista. Pero esto supone entrar en demasiado detalle para el nivel de este texto.

5.2.7. Lectura de listas por teclado

Hasta el momento hemos aprendido a construir listas de diferentes modos, pero nada hemos dicho acerca de cómo leer listas desde el teclado. La función que lee de teclado es *input*, ¿funcionará también con listas?

```
pide_lista.py
lista = input('Dame una lista: ')
print(lista)
```

```
Dame una lista: [1, 2, 3]
[1, 2, 3]
```

¿Ha funcionado? No. Lo que se ha leído es una cadena, no una lista. Podemos cerciorarnos accediendo a su primer elemento: si fuera una lista, valdría 1 y si fuera una cadena, `'['`; o inquiriendo por su longitud: si fuera una lista valdría 3 y si fuera una cadena valdría 9:

```
pide_lista.py
lista = input('Dame una lista: ')
print(lista)
print(lista[0])
print(len(lista))
```

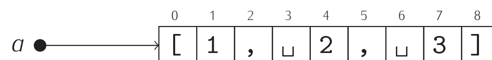
```
Dame una lista: [1, 2, 3]
[1, 2, 3]
[
9
```

De todos modos, era previsible, pues ya dijimos en su momento que *input* devolvía una cadena. Cuando queríamos obtener, por ejemplo, un entero, «encerrábamos» la llamada a *input* con una llamada a la función *int* y cuando queríamos un flotante, con *float*. ¿Habrá alguna función similar para obtener listas? Si queremos una lista, lo lógico sería utilizar una llamada a *list*, que en inglés significa *lista*:

```
pide_lista.py
lista = list(input('Dame una lista: '))
print(lista)
```

```
Dame una lista: [1, 2, 3]
['[', '1', ',', ' ', '2', ',', ' ', '3', ']']
```

¡Oh, oh! Tenemos una lista, sí, pero no la que esperábamos:



La función *list* devuelve una lista a partir de una cadena, pero cada elemento de la lista es un carácter de la cadena (por ejemplo, el `2` que ocupa la posición de índice 4 no es el entero 2, sino el carácter `2`). No se interpreta, pues, como hubiéramos deseado, es decir, como esta lista de números enteros:



Para leer listas deberemos utilizar un método distinto. Lo que haremos es ir leyendo la lista elemento a elemento y construir la lista paso a paso. Este programa, por ejemplo, lee una lista de 5 enteros:

```
lista = []
for i in range(5):
    elemento = int(input('Dame un elemento:'))
```

```
    lista = lista + [elemento]
```

Mejor aún: si usamos *append*, evitaremos que cada concatenación genere una lista nueva copiando los valores de la antigua y añadiendo el elemento recién leído.

```
lista = []
for i in range(5):
    elemento = int(input('Dame␣un␣elemento:'))
    lista.append(elemento)
```

Existe un método alternativo que consiste en crear una lista con 5 celdas y leer después el valor de cada una:

```
lista = [0] * 5
for i in range(5):
    lista[i] = int(input('Dame␣un␣elemento:'))

print(lista)
```

```
Dame un elemento:1↵
Dame un elemento:2↵
Dame un elemento:3↵
Dame un elemento:4↵
Dame un elemento:5↵
[1, 2, 3, 4, 5]
```

Evaluación de expresiones Python en cadenas

Hemos aprendido a leer enteros, flotantes y cadenas con *input*, pero esa función no resulta útil para leer listas. Sin embargo, si evaluamos la cadena como una expresión Python, el intérprete no tendrá mayor problema. La función *eval* evalúa el contenido de una cadena como una expresión Python. Estudia este ejemplo:

```
s = input('Dame␣un␣número:␣')
a = eval(s)
print(a)
s = input('Dame␣una␣cadena:␣')
b = eval(s)
print(b)
s = input('Dame␣una␣lista:␣')
c = eval(s)
print(c)
```

```
Dame un número: 4↵
4
Dame una cadena: 'cadena'↵
cadena
Dame una lista: [1,␣2,␣3]↵
[1, 2, 3]
```

Esta forma de obtener datos del tipo apropiado presenta un gran inconveniente: el usuario de tus programas ha de saber programar en Python, ya que las expresiones deben seguir las reglas sintácticas propias del lenguaje de programación, y eso no es razonable. De todos modos, *eval* puede resultarte de utilidad mientras desarrolles borradores de los programas que diseñes y manejen listas.

Supongamos que deseamos leer una lista de enteros positivos cuya longitud es desconocida. ¿Cómo hacerlo? Podemos ir leyendo números y añadiéndolos a la lista hasta que nos introduzcan un número negativo. El número negativo indicará que hemos finalizado, pero no se añadirá a la lista:

```
lista = []
```

```
número = int(input('Dame␣un␣número:␣'))
while número >= 0:
    lista.append(número)
    número = int(input('Dame␣un␣número:␣'))
```

▶ 221 Diseña un programa que lea una lista de 10 enteros, pero asegurándose de que todos los números introducidos por el usuario son positivos. Cuando un número sea negativo, lo indicaremos con un mensaje y permitiremos al usuario repetir el intento cuantas veces sea preciso.

▶ 222 Diseña un programa que lea una cadena y muestre por pantalla una lista con todas sus palabras en minúsculas. La lista devuelta no debe contener palabras repetidas.

Por ejemplo, ante la cadena

`'Una␣frase␣formada␣con␣palabras.␣␣Otra␣frase␣con␣otras␣palabras.'`,

el programa mostrará la lista

`['una', 'frase', 'formada', 'con', 'palabras', 'otra', 'otras']`.

Observa que en la lista no aparece dos veces la palabra «frase», aunque sí aparecía dos veces en la cadena leída.

5.2.8. Borrado de elementos de una lista

También podemos eliminar elementos de una lista. Para ello utilizamos la sentencia **del** (abreviatura de «delete», que en inglés significa *borrar*). Debes indicar qué elemento deseas eliminar inmediatamente después de la palabra **del**:

```
>>> a = [1, 2, 3]↵
```




```
>>> a = [1, 2, 3]↵
>>> del a[1]↵
>>> a↵
[1, 3]
```



La sentencia **del** *no* produce una copia de la lista sin la celda borrada, sino que modifica directamente la lista sobre la que opera. Fíjate en qué efecto produce si dos variables apuntan a la misma lista:

```
>>> a = [1, 2, 3]↵
>>> b = a↵
>>> del a[1]↵
>>> a↵
[1, 3]
>>> b↵
[1, 3]
```

El borrado de elementos de una lista es peligroso cuando se mezcla con el recorrido de las mismas. Veámoslo con un ejemplo. Hagamos un programa que elimina los elementos negativos de una lista.

```
solo_positivos.py
a = [1, 2, -1, -4, 5, -2]
```

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93
Índice

Las cadenas son inmutables (y III)

Recuerda que las cadenas son inmutables. Esta propiedad también afecta a la posibilidad de borrar elementos de una cadena:

```
>>> a = 'Hola'
>>> del a[1]
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: 'str' object doesn't support item deletion
```

```

for i in a:
    if i < 0:
        del i

print(a)
```

¡Mal! Estamos usando **del** sobre un escalar (*i*), no sobre un elemento indexado de la lista (que, en todo caso, sería *a*[*i*]). Este es un error típico de principiante. La sentencia **del** no se usa así. Vamos con otra versión:

```
solo_positivos.py
a = [1, 2, -1, -4, 5, -2]

for i in range(0, len(a)):
    if a[i] < 0:
        del a[i]

print(a)
```

Ahora sí usamos correctamente la sentencia **del**, pero hay otro problema. Ejecutemos el programa:

```
Traceback (most recent call last):
  File "solo_positivos.py", line 4, in <module>
    if a[i] < 0:
IndexError: list index out of range
```

El mensaje de error nos dice que tratamos de acceder a un elemento con índice fuera del rango de índices válidos. ¿Cómo es posible, si la lista tiene 6 elementos y el índice *i* toma valores desde 0 hasta 5? Al eliminar el tercer elemento (que es negativo), la lista ha pasado a tener 5 elementos, es decir, el índice de su último elemento es 4. Pero el bucle «decidió» el rango de índices a recorrer antes de borrarse ese elemento, es decir, cuando la lista tenía el valor 5 como índice del último elemento. Cuando tratamos de acceder a *a*[5], Python detecta que estamos fuera del rango válido. Es necesario que el bucle «actualice» el valor del último índice válido con cada iteración:

```
solo_positivos.py
a = [1, 2, -1, -4, 5, -2]

i = 0
while i < len(a):
    if a[i] < 0:
        del a[i]
    i += 1

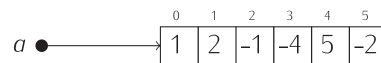
print(a)
```

Ejecutemos el programa:

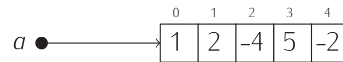
Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

```
[1, 2, -4, 5]
```

¡No ha funcionado! El -4 no ha sido eliminado. ¿Por qué? Inicialmente la lista era:



Al eliminar el elemento *a*[2] de la lista original, *i* valía 2.



Después del borrado, incrementamos *i* y eso hizo que la siguiente iteración considerara el posible borrado de *a*[3], pero en ese instante -4 estaba en *a*[2] (fíjate en la última figura), así que nos lo «saltamos». La solución es sencilla: solo hemos de incrementar *i* en las iteraciones que no producen borrado alguno:

solo_positivos.py

```
a = [1, 2, -1, -4, 5, -2]

i = 0
while i < len(a):
    if a[i] < 0:
        del a[i]
    else:
        i += 1

print(a)
```

Ejecutemos el programa:

```
[1, 2, 5]
```

¡Ahora sí!

▶ 223 ¿Qué sale por pantalla al ejecutar este programa?:

```
a = list(range(0, 5))
del a[1]
del a[1]
print(a)
```

▶ 224 Diseña un programa que elimine de una lista todos los elementos de *índice* par y muestre por pantalla el resultado.

(Ejemplo: si trabaja con la lista [1, 2, 1, 5, 0, 3], esta pasará a ser [2, 5, 3]).

▶ 225 Diseña un programa que elimine de una lista todos los elementos de *valor* par y muestre por pantalla el resultado.

(Ejemplo: si trabaja con la lista [1, -2, 1, -5, 0, 3], esta pasará a ser [1, 1, -5, 3]).

▶ 226 A nuestro programador novato se le ha ocurrido esta otra forma de eliminar el elemento de índice *i* de una lista *a*:

```
a = a[:i] + a[i+1:]
```

¿Funciona? Si no es así, ¿por qué? Y si funciona correctamente, ¿qué diferencia hay con respecto a usar **del** *a*[*i*]?

La sentencia **del** también funciona sobre cortes:

```
>>> a = [1, 2, 3, 4, 5, 6]
>>> del a[2:4]
>>> a
[1, 2, 5, 6]
```

5.2.9. Pertenencia de un elemento a una lista

Diseñemos un programa que, dados un elemento y una lista, nos diga si el elemento pertenece o no a la lista mostrando en pantalla el mensaje «`Pertenece`» o «`No pertenece`» en función del resultado.

```
pertenencia.py
elemento = 5
lista = [1, 4, 5, 1, 3, 8]

pertenece = False
for i in lista:
    if elemento == i:
        pertenece = True
        break

if pertenece:
    print('Pertenece')
else:
    print('No␣pertenece')
```

▶ 227 ¿Por qué este otro programa es erróneo?

```
ϟ pertenencia.py
elemento = 5
lista = [1, 4, 5, 1, 3, 8]

for i in lista:
    if elemento == i:
        pertenece = True
    else:
        pertenece = False

if pertenece:
    print('Pertenece')
else:
    print('No␣pertenece')
```

La pregunta de si un elemento pertenece o no a una lista es tan frecuente que Python nos proporciona un operador predefinido que hace eso mismo. El operador es binario y se denota con la palabra **in** (que en inglés significa «en» o «pertenece a»). El operador **in** recibe un elemento por su parte izquierda y una lista por su parte derecha y devuelve cierto o falso. Un programa que necesita determinar si un elemento pertenece o no a una lista y actuar en consecuencia puede hacerlo así:

```
conjunto.py
conjunto = [1, 2, 3]
elemento = int(input('Dame␣un␣número:␣'))
if not elemento in conjunto:
    conjunto.append(elemento)
print(conjunto)
```

O, equivalentemente:

```
conjunto.py
conjunto = [1, 2, 3]
elemento = int(input('Dame␣un␣número:␣'))
if elemento not in conjunto:
    conjunto.append(elemento)
```

```
print(conjunto)
```

El operador «not in» es el operador in negado.

▶ 228 ¿Qué hace este programa?

```
letra = input('Dame␣una␣letra:␣')
if len(letra) == 1 and ('a'<=letra<='z' or letra in ['á','é','í','ó','ú','ü','ñ']):
    print(letra, 'es␣una␣letra␣minúscula')
```

▶ 229 ¿Qué hace este programa?

```
letra = input('Dame␣una␣letra:␣')
if len(letra) == 1 and ('a'<= letra <='z' or letra in 'áéíóúüñ'):
    print(letra, 'es␣una␣letra␣minúscula')
```

Ya te hemos dicho que Python ofrece funcionalidades similares entre tipos de datos similares. Si el operador in funciona con listas, ¿funcionará con cadenas, que también son secuencias? Sí. El operador in comprueba si una cadena forma parte o no de otra[4]:

```
>>> 'a' in 'cadena'↵
True
>>> 'ade' in 'cadena'↵
True
>>> 'ada' in 'cadena'↵
False
```

Mmmm... ¿podemos entonces utilizar el operador in para comprobar si una lista está contenida en otra?

```
>>> [2, 7] in [1, 2, 7, 4]↵
False
```

No. El operador in, aplicado a listas, solo permite determinar si su operando izquierdo *es un elemento* de la lista. Fíjate en este otro ejemplo:

```
>>> [2, 7] in [1, [2, 7], 4]↵
True
```

En este caso, la lista [2, 7] es el segundo elemento de la lista [1, [2, 7], 4]. ¿Cómo? ¿Un elemento de una lista puede ser otra lista? Sí, ya te dijimos que una lista era una secuencia de valores de *cualquier tipo*. Volveremos sobre este punto cuando estudiemos matrices.

5.2.10. Ordenación de una lista

En este apartado nos ocuparemos de un problema clásico: ordenar (de menor a mayor) los elementos de una lista de valores. La ordenación es muy útil en infinidad de aplicaciones, así que se ha puesto mucho empeño en estudiar algoritmos de ordenación eficientes. De momento estudiaremos únicamente un método muy sencillo (e ineficiente): el *método de la burbuja*. Trataremos de entender bien en qué consiste mediante un ejemplo. Supongamos que deseamos ordenar (de menor a mayor) la lista [2, 26, 4, 3, 1], es decir, hacer que pase a ser [1, 2, 3, 4, 26]. Se procede del siguiente modo:

- Empezamos por comparar los dos primeros elementos (*a*[0] y *a*[1]). Si están ordenados, los dejamos tal cual; si no, los intercambiamos. En nuestro caso ya están ordenados.

0	1	2	3	4
2	26	4	3	1

[4] Este comportamiento solo se da desde la versión 2.3 de Python. Versiones anteriores solo aceptaban que, si ambos operandos eran cadenas, el operador izquierdo fuera de longitud 1.

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

- Ahora comparamos los dos siguientes (*a*[1] y *a*[2]) y hacemos lo mismo.

0	1	2	3	4
2	26	4	3	1

En este caso no están ordenados, así que los intercambiamos y la lista queda así:

0	1	2	3	4
2	4 ⟷	26	3	1

- Ahora comparamos los dos siguientes (*a*[2] y *a*[3]) y hacemos lo mismo.

0	1	2	3	4
2	4	26	3	1

En este caso tampoco están ordenados, así que los intercambiamos y la lista queda así:

0	1	2	3	4
2	4	3 ⟷	26	1

- Ahora comparamos los dos siguientes (*a*[3] y *a*[4]), que son los últimos.

0	1	2	3	4
2	4	3	26	1

En este caso tampoco están ordenados, así que los intercambiamos y la lista queda así:

0	1	2	3	4
2	4	3	1 ⟷	26

La lista aún no está ordenada, pero fíjate en qué ha ocurrido con el elemento más grande de la lista: ya está a la derecha del todo, que es el lugar que le corresponde definitivamente.

0	1	2	3	4
2	4	3	1	26 ☺

Desde que hemos examinado ese valor, cada paso del procedimiento lo ha movido una posición a la derecha. De hecho, el nombre de este procedimiento de ordenación (método de la burbuja) toma el nombre del comportamiento que hemos observado. Es como si las burbujas en un medio líquido subieran hacia la superficie del mismo: las más grandes alcanzarán el nivel más próximo a la superficie y lo harán rápidamente.

Ahora solo es preciso ordenar los 4 primeros elementos de la lista, así que aplicamos el mismo procedimiento a esa «sublista»:

- Empezamos por comparar los dos primeros elementos (*a*[0] y *a*[1]). Si están ordenados, los dejamos tal cual; si no, los intercambiamos.

0	1	2	3	4
2	4	3	1	26 ☺

La importancia de ordenar rápidamente

Ordenar el contenido de una lista es un problema importante porque se plantea en numerosos campos de aplicación de la programación: la propia palabra «ordenador» lo pone de manifiesto. Ordenar es, quizá, el problema más estudiado y para el que existe mayor número de soluciones diferentes, cada una con sus ventajas e inconvenientes o especialmente adaptada para tratar casos particulares.

Podemos citar aquí a Donald E. Knuth en el tercer volumen («Sorting and searching») de «The art of computer programming», un texto clásico de programación: «*Los fabricantes de ordenadores de los años 60 estimaron que más del 25 por ciento del tiempo de ejecución en sus ordenadores se dedicaba a ordenar cuando consideraban al conjunto de sus clientes. De hecho, había muchas instalaciones en las que la tarea de ordenar era responsable de más de la mitad del tiempo de computación. De estas estadísticas podemos concluir que (i) la ordenación cuenta con muchas aplicaciones importantes, (ii) mucha gente ordena cuando no debiera, o (iii) se usan comúnmente algoritmos de ordenación ineficientes.*»

En nuestro caso ya están ordenados.

- Ahora comparamos los dos siguientes (a[1] y a[2]) y hacemos lo mismo.

0	1	2	3	4
2	4	3	1	26

En este caso no están ordenados, así que los intercambiamos y la lista queda así:

0	1	2	3	4
2	3 ⟷	4	1	26

- Ahora comparamos los dos siguientes (a[2] y a[3]) y hacemos lo mismo.

0	1	2	3	4
2	3	4	1	26

En este caso tampoco están ordenados, así que los intercambiamos y la lista queda así:

0	1	2	3	4
2	3	1 ⟷	4	26

Ahora resulta que el segundo mayor elemento ya está en su posición definitiva. Parece que cada vez que recorremos la lista, al menos un elemento se ubica en su posición definitiva: el mayor de los que aún estaban por ordenar.

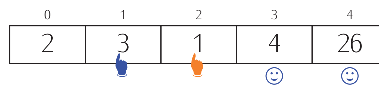
A ver qué ocurre en el siguiente recorrido (que se limitará a la «sublista» de los tres primeros elementos, pues los otros dos ya están bien puestos):

- Empezamos por comparar los dos primeros elementos (a[0] y a[1]). Si están ordenados, los dejamos tal cual; si no, los intercambiamos.

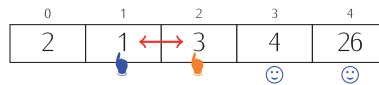
0	1	2	3	4
2	3	1	4	26

En nuestro caso ya están ordenados.

- Ahora comparamos los dos siguientes (a[1] y a[2]) y hacemos lo mismo.



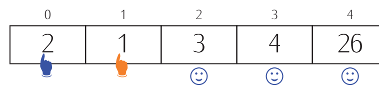
En este caso no están ordenados, así que los intercambiamos y la lista queda así:



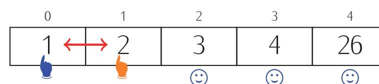
Ya tenemos en su lugar los tres últimos elementos.

Parece que nuestra hipótesis es cierta. Aún nos falta un poco para acabar:

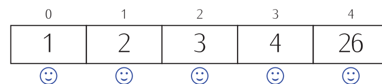
- Comparamos los dos primeros elementos (a[0] y a[1]). Si están ordenados, los dejamos tal cual; si no, los intercambiamos.



No están ordenados, así que los intercambiamos. La lista queda, finalmente, así:



¡Perfecto!: la lista ha quedado completamente ordenada.



Recapitulemos: para ordenar una lista de n elementos hemos de hacer $n-1$ pasadas. En cada pasada conseguimos poner al menos un elemento en su posición: el mayor. (Hacen falta $n-1$ y no n porque la última pasada nos pone *dos* elementos en su sitio: el mayor va a la segunda posición y el menor se queda en el único sitio que queda: la primera celda de la lista). Intentemos codificar esa idea en Python:

burbuja.py
```
lista = [2, 26, 4, 3, 1]

for i in range(1, len(lista)): # Bucle que hace len(lista)-1 pasadas.
    hacer una pasada

print(lista)
```

¿En qué consiste la i-ésima pasada? En explorar todos los pares de celdas contiguas, desde el primero hasta el último. En cada paso comparamos un par de elementos:

burbuja.py
```
lista = [2, 26, 4, 3, 1]

for i in range(1, len(lista)):
    for j in range(0, len(lista)-i):
```

```
        comparar lista[j] y lista[j+1] y, si procede, intercambiarlos

print(lista)
```

Lo que queda debería ser fácil:

```
burbuja.py
lista = [2, 26, 4, 3, 1]

for i in range(1, len(lista)):
    for j in range(0, len(lista)-i):
        if lista[j] > lista[j+1]:
            elemento = lista[j]
            lista[j] = lista[j+1]
            lista[j+1] = elemento

print(lista)
```

¡Buf! ¿Estará bien? He aquí el resultado de ejecutar el programa:

```
[1, 2, 3, 4, 26]
```

¡Sí! Pero, ¿estará bien con seguridad? Para tener una certeza mayor, vamos a modificar el programa para que nos diga por pantalla qué hace en cada instante:

```
burbuja.py
lista = [2, 26, 4, 3, 1]

for i in range(1, len(lista)):
    print('Pasada', i)
    for j in range(0, len(lista)-i):
        print('␣␣Comparación␣de␣lista[{0}]␣y␣lista[{1}]'.format(j, j+1))
        if lista[j] > lista[j+1]:
            elemento = lista[j]
            lista[j] = lista[j+1]
            lista[j+1] = elemento
            print('␣␣Se␣intercambian')
        print('␣␣Estado␣actual␣de␣la␣lista', lista)

print(lista)
```

Probemos de nuevo:

```
Pasada 1
  Comparación de lista[0] y lista[1]
  Estado actual de la lista [2, 26, 4, 3, 1]
  Comparación de lista[1] y lista[2]
  Se intercambian
  Estado actual de la lista [2, 4, 26, 3, 1]
  Comparación de lista[2] y lista[3]
  Se intercambian
  Estado actual de la lista [2, 4, 3, 26, 1]
  Comparación de lista[3] y lista[4]
  Se intercambian
  Estado actual de la lista [2, 4, 3, 1, 26]
Pasada 2
  Comparación de lista[0] y lista[1]
  Estado actual de la lista [2, 4, 3, 1, 26]
  Comparación de lista[1] y lista[2]
  Se intercambian
  Estado actual de la lista [2, 3, 4, 1, 26]
  Comparación de lista[2] y lista[3]
  Se intercambian
  Estado actual de la lista [2, 3, 1, 4, 26]
Pasada 3
```

```
  Comparación de lista[0] y lista[1]
  Estado actual de la lista [2, 3, 1, 4, 26]
  Comparación de lista[1] y lista[2]
  Se intercambian
  Estado actual de la lista [2, 1, 3, 4, 26]
Pasada 4
  Comparación de lista[0] y lista[1]
  Se intercambian
  Estado actual de la lista [1, 2, 3, 4, 26]
[1, 2, 3, 4, 26]
```

Bueno, seguros de que esté bien no estamos, pero al menos sí parece hacer lo que toca. Ya podemos eliminar las sentencias *print* que hemos introducido en el programa para hacer esta traza automática. Mostrar los mensajes que informan de por dónde pasa el flujo de ejecución de un programa y del contenido de algunas de sus variables es un truco frecuentemente utilizado por los programadores para ver si un programa hace lo que debe y, cuando el programa tiene errores, detectarlos y corregirlos. Por supuesto, una vez nos hemos asegurado de que el programa funciona, hemos de eliminar las sentencias adicionales.

▶ 230 ¿Qué ocurrirá si sustituimos la primera línea de **burbuja.py** por esta otra?:

```
lista = ['Pepe', 'Juan', 'María', 'Ana', 'Luis', 'Pedro']
```

Depuración y corrección de programas

Es muy frecuente que un programa no se escriba bien a la primera. Por regla general, gran parte del tiempo de programación se dedica a buscar y corregir errores. Esta actividad se denomina *depurar* el código (en inglés, «debugging», que significa «desinsectar»). Existen herramientas de ayuda a la depuración: los *depuradores* (en inglés, *debuggers*). Un depurador permite ejecutar paso a paso un programa bajo el control del programador y consultar en cualquier instante el valor de las variables.

Pero con la ayuda de un buen depurador nunca podemos estar seguros de que un programa esté bien. Cuando un programa aborta su ejecución o deja colgado al ordenador es evidente que hay un error, pero, ¿cómo podemos estar seguros de que un programa que, de momento, parece funcionar bien, lo hará siempre? ¿Y si tiene un error tan sutil que solo se manifiesta ante una entrada muy particular? Por extraña que sea esa entrada, cabe la posibilidad de que el programa se enfrente a ella durante su utilización por parte de los usuarios. Y cuando eso ocurra, el programa abortará su ejecución o, peor aún, ofrecerá resultados mal calculados como si fueran buenos. Asusta pensar que de ese programa puedan depender vidas humanas, cosa que ocurre en no pocos casos (programas para el cálculo de estructuras en edificaciones, para el lanzamiento y guiado de naves espaciales, para el control de centrales nucleares, etc.).

Existe una serie de técnicas matemáticas para *demostrar* que un programa hace lo que se le pide. Bajo ese enfoque, demostrar que un programa es correcto equivale a demostrar un teorema.

5.3. De cadenas a listas y viceversa

En muchas ocasiones nos encontraremos convirtiendo cadenas en listas y viceversa. Python nos ofrece una serie de utilidades que conviene conocer si queremos ahorrarnos muchas horas de programación.

Una acción frecuente consiste en obtener una lista con todas las palabras de una cadena. He aquí cómo puedes hacerlo:

```
>>> 'uno dos tres'.split()
['uno', 'dos', 'tres']
```

En inglés «split» significa «partir». ¿Funcionará con textos «maliciosos», es decir, con espacios en blanco al inicio, al final o repetidos?

```
>>> '   uno  dos tres  '.split()
['uno', 'dos', 'tres']
```

Pickle

Con lo aprendido hasta el momento ya puedes hacer algunos programas interesantes. Puedes ir anotando en una lista ciertos datos de interés, como los apuntes de una cuenta bancaria (serie de flotantes con valor positivo o negativo para ingresos y reintegros, respectivamente). El problema estriba en que tu programa tendría que leer de teclado la lista entera ¡cada vez que se ejecutara! No es una forma natural de funcionar.

Te vamos a enseñar una técnica que te permite guardar una lista en el disco duro y recuperarla cuando quieras. Tu programa podría empezar a ejecutarse leyendo la lista del disco duro y, justo antes de acabar la ejecución, guardar nuevamente la lista en el disco duro.

El módulo *pickle* permite guardar/cargar estructuras de datos Python. Vemos un ejemplo:

guardar.py
```
import pickle

# Creamos una lista ...
lista = [1, 2, 3, 4]
# y ahora la guardamos en un fichero llamado mifichero.mio.
pickle.dump(lista, open('mifichero.mio', 'wb'))
```

Al ejecutar ese programa, se crea un fichero cuyo contenido es la lista. Este otro programa leería la misma lista:

cargar.py
```
import pickle

# Leemos la lista cargándola del fichero mifichero.mio...
lista = pickle.load(open('mifichero.mio', 'rb'))
# y la mostramos por pantalla.
print(lista)
```

Nos hemos anticipado un poco al capítulo dedicado a la gestión de ficheros, pero de este modo te estamos capacitando para que hagas programas que pueden «recordar» información entre diferentes ejecuciones. Si quieres saber más, lee la documentación del módulo *pickle*. ¡Que lo disfrutes!

Sí. Fantástico. ¿Recuerdas los quebraderos de cabeza que supuso contar el número de palabras de una frase? Mira cómo se puede calcular con la ayuda de *split*:

```
>>> len('␣␣␣uno␣␣dos␣tres␣␣'.split())↵
3
```

El método *split* acepta un argumento opcional: el carácter «divisor», que por defecto es el espacio en blanco:

```
>>> 'uno:dos␣tres:cuatro'.split(':')↵
['uno', 'dos␣tres', 'cuatro']
```

▶ 231 En una cadena llamada *texto* disponemos de un texto formado por varias frases. ¿Con qué orden simple puedes contar el número de frases?

▶ 232 En una cadena llamada *texto* disponemos de un texto formado por varias frases. Escribe un programa que determine y muestre el número de palabras de cada frase.

Hay un método que hace lo contrario: une las cadenas de una lista en una sola cadena. Su nombre es *join* (que en inglés significa «unir») y se usa así:

```
>>> '␣'.join(['uno', 'dos', 'tres'])↵
'uno␣dos␣tres'
>>> ':'.join(['uno', 'dos', 'tres'])↵
'uno:dos:tres'
>>> '--'.join(['uno', 'dos', 'tres'])↵
'uno--dos--tres'
```

¿Ves? Se usa una cadena a mano izquierda del punto y se suministra una lista como argumento. El resultado es una cadena formada por los elementos de la lista separados entre sí por la cadena a mano izquierda.

▶ 233 ¿Qué resulta de ejecutar esta sentencia?

```
>>> print(''.join(['uno', 'dos', 'tres']))↵
```

▶ 234 Disponemos de una cadena que contiene una frase cuyas palabras están separadas por un número arbitrario de espacios en blanco. ¿Podrías «estandarizar» la separación de palabras en una sola línea Python? Por estandarizar queremos decir que la cadena no empiece ni acabe con espacios en blanco y que cada palabra se separe de la siguiente por un único espacio en blanco.

Hay, además, una función predefinida que permite convertir una cadena en una lista: *list*. La función *list* devuelve una lista formada por los caracteres individuales de la cadena:

```
>>> list('cadena')↵
['c', 'a', 'd', 'e', 'n', 'a']
```

Los métodos *join* y *split* son insustituibles en la caja de herramientas de un programador Python. Acostúmbrate a utilizarlos.

5.4. Matrices

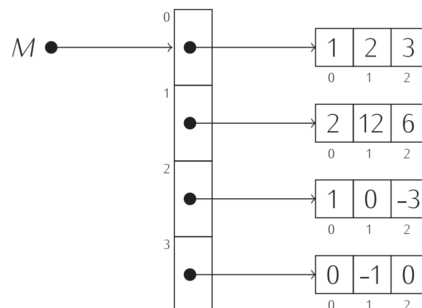
Las matrices son disposiciones bidimensionales de valores. En notación matemática, una matriz se denota encerrando entre paréntesis los valores, que se disponen en *filas* y *columnas*. He aquí una matriz M:

$$M = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 12 & 6 \\ 1 & 0 & -3 \\ 0 & -1 & 0 \end{pmatrix}$$

Esta matriz tiene 4 filas y 3 columnas, lo cual abreviamos diciendo que es una matriz de dimensión 4×3.

Las listas permiten representar series de datos en una sola dimensión. Con una lista de números no se puede representar directamente una matriz, pero sí con una lista de *listas de números*.

```
>>> M = [ [1, 2, 3], [2, 12, 6], [1, 0, -3], [0, -1, 0] ]↵
```



En la notación matemática, el elemento que ocupa la fila i-ésima y la columna j-ésima de una matriz M se representa con $M_{i,j}$. Por ejemplo, el elemento de una matriz que ocupa la celda de la fila 1 y la columna 2 se denota con $M_{1,2}$. Pero si deseamos acceder a ese elemento en la matriz Python M, hemos de tener en cuenta que Python siempre cuenta desde cero, así que la fila tendrá índice 0 y la columna tendrá índice 1:

```
>>> M = [ [1, 2, 3], [2, 12, 6], [1, 0, -3], [0, -1, 0] ]↵
```

```
>>> M[0][1]
2
```

Observa que utilizamos una doble indexación para acceder a elementos de la matriz. ¿Por qué? El primer índice aplicado sobre M devuelve un componente de M, que es una lista:

```
>>> M = [ [1, 2, 3], [2, 12, 6], [1, 0, -3], [0, -1, 0] ]
>>> M[0]
[1, 2, 3]
```

Y el segundo índice accede a un elemento de esa lista, que es un entero:

```
>>> M = [ [1, 2, 3], [2, 12, 6], [1, 0, -3], [0, -1, 0] ]
>>> M[0][0]
1
```

▶ 235 Una matriz nula es aquella que solo contiene ceros. Construye una matriz nula de 5 filas y 5 columnas.

▶ 236 Una matriz identidad es aquella cuyos elementos en la diagonal principal, es decir, accesibles con una expresión de la forma M[i][i], valen uno y el resto valen cero. Construye una matriz identidad de 4 filas y 4 columnas.

▶ 237 ¿Qué resulta de ejecutar este programa?

```
M = [ [1, 0, 0], [0, 1, 0], [0, 0, 1] ]
print(M[-1][0])
print(M[-1][-1])
print('-')
for i in range(0, 3):
    print(M[i])
print('-')
for i in range(0, 3):
    for j in range(0, 3):
        print(M[i][j])
```

▶ 238 ¿Qué resulta de ejecutar este programa?

```
M = [ [1, 0, 0], [0, 1, 0], [0, 0, 1] ]
s = 0.0
for i in range(0, 3):
    for j in range(0, 3):
        s += M[i][j]
print(s / 9)
```

5.4.1. Sobre la creación de matrices

Crear una matriz consiste, pues, en crear una lista de listas. Si deseamos crear una matriz nula (una matriz cuyos componentes sean todos igual a 0) de tamaño 2×2, bastará con escribir:

```
>>> m = [ [0, 0], [0, 0] ]
```

Parece sencillo, pero ¿y si nos piden una matriz nula de 6×6? Tiene 36 componentes y escribirlos explícitamente resulta muy tedioso. ¡Y pensemos en lo inviable de definir así una matriz de dimensión 10×10 o 100×100!

Recuerda que hay una forma de crear listas (vectores) de cualquier tamaño, siempre que tengan el mismo valor, utilizando el operador *:

```
>>> a = [0] * 6
>>> a
[0, 0, 0, 0, 0, 0]
```

Si una matriz es una lista de listas, ¿qué ocurrirá si creamos una lista con 3 duplicados de la lista *a*?

```
>>> a = [0] * 6↵
>>> [a] * 3↵
[[0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0]]
```

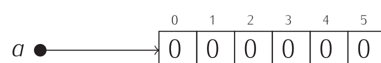
¡Estupendo! Ya tenemos una matriz nula de 3×6. Trabajemos con ella:

```
>>> a = [0] * 6↵
>>> M = [a] * 3↵
>>> M[0][0] = 1↵
>>> print(M)↵
[[1, 0, 0, 0, 0, 0], [1, 0, 0, 0, 0, 0], [1, 0, 0, 0, 0, 0]]
```

¿Qué ha ocurrido? ¡No se ha modificado únicamente el componente 0 de la primera lista, sino *todos* los componentes 0 de todas las listas de la matriz!

Vamos paso a paso. Primero hemos creado *a*:

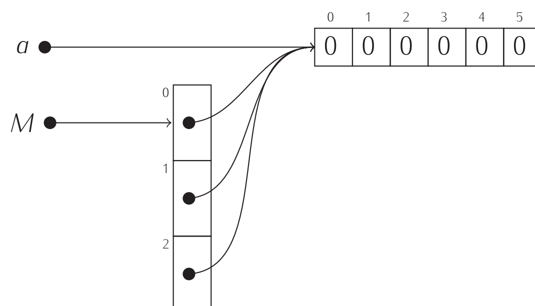
```
>>> a = [0] * 6↵
```



A continuación hemos definido la lista *M* como la copia por triplicado de la lista *a*:

```
>>> a = [0] * 6↵
>>> M = [a] * 3↵
```

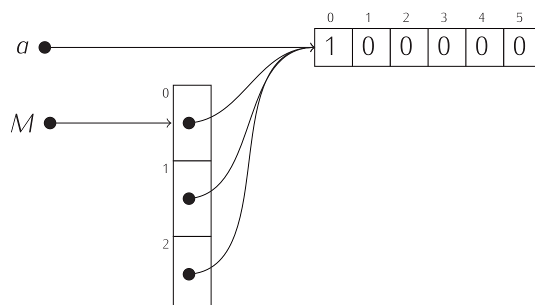
Python nos ha obedecido copiando tres veces... ¡la referencia a dicha lista!:



Y hemos modificado el elemento *M*[0][0] asignándole el valor 1:

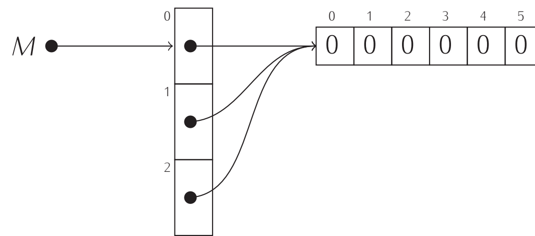
```
>>> a = [0] * 6↵
>>> M = [a] * 3↵
>>> M[0][0] = 1↵
```

así que hemos modificado también *M*[1][0] y *M*[2][0], pues *son el mismo elemento*:



Por la misma razón, tampoco funcionará este modo más directo de crear una matriz:

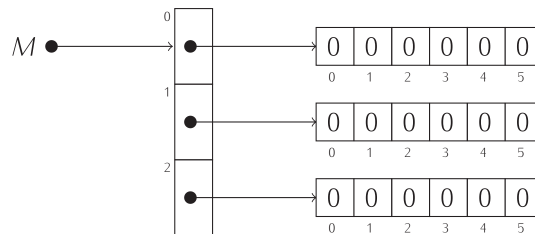
```
>>> M = [ [0] * 6 ] * 3↵
```



Hay que construir matrices con más cuidado, asegurándonos de que cada fila es una lista diferente de las anteriores. Intentémoslo de nuevo:

```
>>> M = []↵
>>> for i in range(3):↵
...     a = [0] * 6↵
...     M.append( a )↵
... ↵
>>> print(M)↵
[[0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0]]
```

La lista creada en la asignación a = [0] * 6 es diferente con cada iteración, así que estamos añadiendo a M una lista nueva cada vez. La memoria queda así:



Al ejecutarse el bucle, se ha construido una fila nueva por cada iteración. A efectos de construcción de la matriz, la ejecución de aquellas sentencias equivale a la de estas:

```
>>> M = []↵
>>> a = [0] * 6↵
>>> M.append( a )↵
>>> a = [0] * 6↵
>>> M.append( a )↵
>>> a = [0] * 6↵
>>> M.append( a )↵
>>> ↵
>>> print(M)↵
[[0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0]]
```

Lo cierto es que no es necesario utilizar la variable auxiliar a:

```
>>> M = []↵
>>> M.append([0] * 6)↵
>>> M.append([0] * 6)↵
>>> M.append([0] * 6)↵
>>> ↵
>>> print(M)↵
[[0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0]]
```

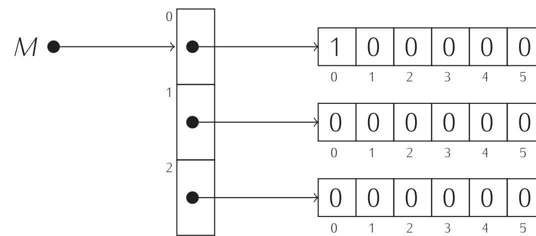
Y con con tanta sentencia repetida, un bucle resulta más elegante:

```
>>> M = []↵
>>> for i in range(3):↵
...     M.append( [0] * 6 )↵
... ↵
>>> print(M)↵
[[0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0]]
```

Observa cómo modificar ahora una celda de una fila no afecta a las demás:

```
>>> M = []↵
>>> for i in range(3):↵
```

```
...     M.append( [0] * 6 )↵
... ↵
>>> M[0][0] = 1↵
>>> print(M)↵
[[1, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0]]
```



---

239 Crea la siguiente matriz utilizando la técnica del bucle descrita anteriormente.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

240 Haz un programa que pida un entero positivo n y almacene en una variable M la matriz identidad de $n \times n$ (la que tiene unos en la diagonal principal y ceros en el resto de celdas).

5.4.2. Lectura de matrices

Si deseamos leer una matriz de tamaño determinado, podemos crear una matriz nula como hemos visto en el apartado anterior y, a continuación, rellenar cada uno de sus componentes:

matrices.py
```
# Pedimos la dimensión de la matriz,
m = int(nput('Dime␣el␣número␣de␣filas:␣'))
n = int(input('Dime␣el␣número␣de␣columnas:␣'))

# Creamos una matriz nula...
M = []
for i in range(m):
    M.append( [0] * n )

# ... y leemos su contenido de teclado
for i in range(m):
    for j in range(n):
      M[i][j] = float(input('Dame␣el␣componente␣({0},{1}):␣'.format(i, j)))
```

5.4.3. ¿Qué dimensión tiene una matriz?

Cuando deseábamos saber cuál era la longitud de una lista utilizábamos la función *len*. ¿Funcionará también sobre matrices? Hagamos la prueba:

```
>>> a = [[1, 0], [0, 1], [0, 0]]↵
>>> len(a)↵
3
```

No funciona correctamente: solo nos devuelve el número de filas (que es el número de componentes de la lista de listas que es la matriz). ¿Cómo averiguar el número de columnas? Fácil:

```
>>> a = [[1, 0], [0, 1], [0, 0]]↲
>>> len(a[0])↲
2
```

5.4.4. Operaciones con matrices

Desarrollemos ahora algunos programas que nos ayuden a efectuar operaciones con matrices como la suma o el producto.

Empecemos por diseñar un programa que sume dos matrices. Recuerda que solo es posible sumar matrices con la misma dimensión, así que solicitaremos una sola vez el número de filas y columnas:

suma_matrices.py

```
# Pedimos la dimensión de las matrices,
m = int(input('Dime␣el␣número␣de␣filas:␣'))
n = int(input('Dime␣el␣número␣de␣columnas:␣'))

# Creamos dos matrices nulas...
A = []
for i in range(m):
    A.append( [0] * n )

B = []
for i in range(m):
    B.append( [0] * n )

# ... y leemos sus contenidos de teclado.
print('Lectura␣de␣la␣matriz␣A')
for i in range(m):
    for j in range(n):
        A[i][j] = float(input('Componente␣({0},{1}):␣'.format(i, j)))

print('Lectura␣de␣la␣matriz␣B')
for i in range(m):
    for j in range(n):
        B[i][j] = float(input('Componente␣({0},{1}):␣'.format(i, j)))
```

Hemos de tener claro cómo se calcula $C = A + B$. Si la dimensión de A y de B es $m \times n$, la matriz resultante será de esa misma dimensión, y su elemento de coordenadas (i, j), es decir, $C_{i,j}$, se calcula así:

$$C_{i,j} = A_{i,j} + B_{i,j},$$

para $1 \leq i \leq m$ y $1 \leq j \leq n$. Recuerda que la convención adoptada en la notación matemática hace que los índices de las matrices empiecen en 1, pero que en Python todo empieza en 0. Codifiquemos ese cálculo en Python.

suma_matrices.py

```
m = int(input('Dime␣el␣número␣de␣filas:␣'))
n = int(input('Dime␣el␣número␣de␣columnas:␣'))

# Creamos dos matrices nulas...
A = []
for i in range(m):
    A.append( [0] * n )

B = []
for i in range(m):
    B.append( [0] * n )

# ... y leemos sus contenidos de teclado.
print('Lectura␣de␣la␣matriz␣A')
```

```
for i in range(m):
    for j in range(n):
        A[i][j] = float(input('Componente␣({0},{1}):␣'.format(i, j)))

print('Lectura␣de␣la␣matriz␣B')
for i in range(m):
    for j in range(n):
        B[i][j] = float(input('Componente␣({0},{1}):␣'.format(i, j)))

# Construimos otra matriz nula para albergar el resultado.
C = []
for i in range(m):
    C.append( [0] * n )

# Empieza el cálculo de la suma.
for i in range(m):
    for j in range(n):
        C[i][j] = A[i][j] + B[i][j]

# Y mostramos el resultado por pantalla
print('Suma:')
for i in range(m):
    for j in range(n):
        print(C[i][j], end='␣')
    print()
```

```
Dime el número de filas: 2↵
Dime el número de columnas: 2↵
Lectura de la matriz A
Componente (0,0): 1↵
Componente (0,1): 2↵
Componente (1,0): 3↵
Componente (1,1): 4↵
Lectura de la matriz B
Componente (0,0): 10↵
Componente (0,1): 20↵
Componente (1,0): 30↵
Componente (1,1): 40↵
Suma:
11.0 22.0
33.0 44.0
```

▶ 241 Diseña un programa que lea dos matrices y calcule la diferencia entre la primera y la segunda.

▶ 242 Diseña un programa que lea una matriz y un número y devuelva una nueva matriz: la que resulta de multiplicar la matriz por el número. (El producto de un número por una matriz es la matriz que resulta de multiplicar cada elemento por dicho número).

Multiplicar matrices es un poco más difícil que sumarlas (y, por descontado, el operador `*` no calcula el producto de matrices). Una matriz A de dimensión $p \times q$ se puede multiplicar por otra matriz B si esta es de dimensión $q \times r$, es decir, si el número de columnas de la primera es igual al número de filas de la segunda. Hemos de pedir, pues, el número de filas y columnas de la primera matriz y solo el número de columnas de la segunda.

multiplica_matrices.py

```
# Pedimos la dimensión de la primera matriz y el número de columnas de la segunda.
p = int(input('Dime␣el␣número␣de␣filas␣de␣A:␣'))
q = int(input('Dime␣el␣número␣de␣columnas␣de␣A␣(y␣filas␣de␣B):␣'))
r = int(input('Dime␣el␣número␣de␣columnas␣de␣B:␣'))

# Creamos dos matrices nulas...
```

```
A = []
for i in range(p):
    A.append( [0] * q )

B = []
for i in range(q):
    B.append( [0] * r )

# ... y leemos sus contenidos de teclado.
print('Lectura de la matriz A')
for i in range(p):
    for j in range(q):
        A[i][j] = float(input('Componente ({0},{1}): '.format(i, j)))

print('Lectura de la matriz B')
for i in range(q):
    for j in range(r):
        B[i][j] = float(input('Componente ({0},{1}): '.format(i, j)))
```

Sigamos. La matriz resultante del producto es de dimensión $p \times r$:

```
multiplica_matrices.py
# Creamos una matriz nula más para el resultado...
C = []
for i in range(p):
    C.append( [0] * r )
```

El elemento de coordenadas $C_{i,j}$ se calcula así:

$$C_{i,j} = \sum_{k=1}^{q} A_{i,k} \cdot B_{k,j},$$

para $1 \leq i \leq p$ y $1 \leq j \leq r$.

```
multiplica_matrices.py
# Pedimos la dimensión de la primera matriz y el número de columnas de la segunda.
p = int(input('Dime el número de filas de A: '))
q = int(input('Dime el número de columnas de A (y filas de B): '))
r = int(input('Dime el número de columnas de B: '))

# Creamos dos matrices nulas...
A = []
for i in range(p):
    A.append( [0] * q )

B = []
for i in range(q):
    B.append( [0] * r )

# ... y leemos sus contenidos de teclado.
print('Lectura de la matriz A')
for i in range(p):
    for j in range(q):
        A[i][j] = float(input('Componente ({0},{1}): '.format(i, j)))

print('Lectura de la matriz B')
for i in range(q):
    for j in range(r):
        B[i][j] = float(input('Componente ({0},{1}): '.format(i, j)))

# Creamos una matriz nula más para el resultado...
```

```
27 C = []
28 for i in range(p):
29     C.append( [0] * r )
30
31 # Y efectuamos el cálculo del producto.
32 for i in range(p):
33     for j in range(r):
34         for k in range(q):
35             C[i][j] += A[i][k] * B[k][j]
```

¿Complicado? No tanto: a fin de cuentas las líneas 34–35 corresponden al cálculo de un sumatorio, algo que hemos codificado en Python una y otra vez.

Solo falta mostrar el resultado por pantalla, pero ya hemos visto cómo se hace. Completa tú el programa.

Otros usos de las matrices

De momento solo hemos discutido aplicaciones numéricas de las matrices, pero son útiles en muchos otros campos. Por ejemplo, muchos juegos de ordenador representan informaciones mediante matrices:

- El tablero de tres en raya es una matriz de 3×3 en el que cada casilla está vacía o contiene la ficha de un jugador, así que podríamos codificar con el valor 0 el que esté vacía, con el valor 1 el que tenga una ficha de un jugador y con un 2 el que tenga una ficha del otro jugador.
- Un tablero de ajedrez es una matriz de 8×8 en el que cada casilla está vacía o contiene una pieza. ¿Cómo las codificarías?
- El tablero del juego del buscaminas es una matriz. En cada celda se codifica si hay bomba o no y si el usuario la ha descubierto ya o no.
- ...

Las cámaras de video digitales permiten recoger imágenes, cada una de las cuales no es más que una matriz de valores. Si la imagen es en blanco y negro, cada valor es un número que representa la intensidad de brillo en ese punto; si la imagen es en color, cada casilla contiene tres valores: la intensidad de la componente roja, la de la componente verde y la de la componente azul. Los sistemas de visión artificial aplican transformaciones a esas matrices y las analizan para tratar de identificar en ellas determinados objetos.

▶ 243 La traspuesta de una matriz A de dimensión $m \times n$ es una matriz A^T de dimensión $n \times m$ tal que $A^T_{i,j} = A_{j,i}$. Por ejemplo, si

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 12 & 6 \\ 1 & 0 & -3 \\ 10 & -1 & 0 \end{pmatrix}$$

entonces:

$$A^T = \begin{pmatrix} 1 & 2 & 1 & 10 \\ 2 & 12 & 0 & -1 \\ 3 & 6 & -3 & 0 \end{pmatrix}$$

Diseña un programa que lea una matriz y muestre su traspuesta.

▶ 244 Diseña un programa tal que lea una matriz A de dimensión $m \times n$ y muestre un vector v de talla n tal que

$$v_j = \sum_{i=1}^{m} A_{i,j},$$

para j entre 1 y n.

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

▶ 245 Diseña un programa que lea una matriz A de dimensión $m \times n$ y muestre un vector v de talla $\min(n, m)$ tal que

$$v_i = \sum_{j=1}^{i} \sum_{k=1}^{i} A_{j,k},$$

para i entre 1 y $\min(n, m)$.

▶ 246 Diseña un programa que, dada una matriz, determine si la suma de los elementos de cualquiera de sus filas es igual a la suma de los elementos de cualquiera de sus columnas.

▶ 247 Una matriz cuadrada es triangular superior si todos los elementos por debajo de la diagonal principal son nulos. Por ejemplo, esta matriz es triangular superior:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 0 & 12 & 6 \\ 0 & 0 & -3 \end{pmatrix}$$

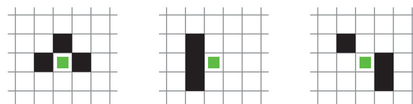
Diseña un programa que diga si una matriz es o no es triangular superior.

5.4.5. El juego de la vida

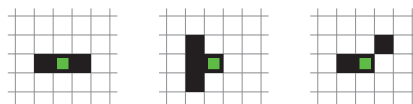
El juego de la vida es un juego sin jugadores. Se trata de colocar una serie de fichas en un tablero y dejar que evolucionen siguiendo unas reglas extremadamente simples. Lo curioso es que esas reglas dan origen a una gran complejidad que hace apasionante la mera observación de la evolución de las fichas en el tablero (hay gustos para todo).

En el juego original se utiliza un tablero (una matriz) con infinitas filas y columnas. Como disponer de una matriz de dimensión infinita en un programa es imposible, supondremos que presenta dimensión $m \times n$, donde m y n son valores escogidos por nosotros. Cada celda del tablero contiene una célula que puede estar viva o muerta. Representaremos las células vivas con su casilla de color negro y las células muertas con la celda en blanco. Cada casilla del tablero cuenta con ocho celdas vecinas. El mundo del juego de la vida está gobernado por un reloj que marca una serie de pulsos con los que mueren y nacen células. Cuándo nace y cuándo muere una célula solo depende de cuántas células vecinas están vivas. He aquí las reglas:

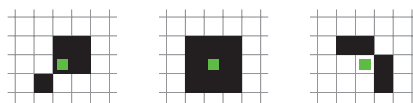
1) Regla del nacimiento. Una célula muerta resucita si tiene exactamente tres vecinos vivos. En estas figuras te señalamos celdas muertas que pasan a estar vivas con el siguiente pulso:



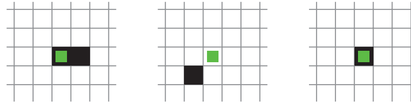
2) Regla de la supervivencia. Una celda viva permanece viva si tiene dos o tres vecinos. Aquí te señalamos células que ahora están vivas y permanecerán así tras el siguiente pulso:



3) Regla de la superpoblación. Una célula muere o permanece muerta si tiene cuatro o más vecinos. Estas figuras muestran células que ahora están vivas o muertas y estarán muertas tras el siguiente pulso:



4) Regla del aislamiento. Una célula muere o permanece muerta si tiene menos de dos vecinos. En estas figuras te señalamos células que ahora están vivas o muertas y estarán muertas tras el siguiente pulso:



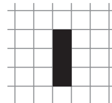
Vamos a hacer un programa que muestre la evolución del juego de la vida durante una serie de pulsos de reloj. Empezaremos con un prototipo que nos muestra la evolución del tablero en el terminal.

Necesitamos representar de algún modo nuestro «universo»: el tablero de celdas. Evidentemente, se trata de una matriz. ¿De qué dimensión? La que queramos. Usaremos dos variables: *filas* y *columnas* para la dimensión y una matriz de valores lógicos para representar el tablero. Inicializaremos el tablero con valores False y, para hacer pruebas, supondremos que la matriz es de 10×10:

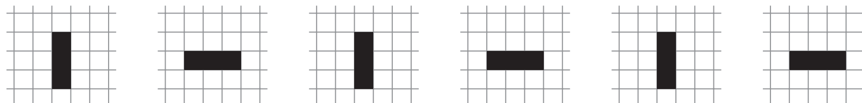
```
vida.py
filas = 10
columnas = 10

tablero = []
for i in range(filas):
    tablero.append([False]*columnas)
```

Ahora deberíamos inicializar el universo ubicando algunas células vivas. De lo contrario, nunca aparecerá «vida» en el juego. Un patrón sencillo y a la vez interesante es este:



Fíjate en qué ocurre tras unos pocos pulsos de actividad:



Es lo que denominamos un oscilador: alterna entre dos o más configuraciones.

```
vida.py
tablero[4][5] = True
tablero[5][5] = True
tablero[6][5] = True
```

Ahora deberíamos representar el tablero de juego en pantalla. Usaremos de momento el terminal de texto: un punto representará una célula muerta y un asterisco representará una célula viva.

```
vida.py
filas = 10
columnas = 10

tablero = []
for i in range(filas):
    tablero.append([False]*columnas)

tablero[4][5] = True
```

```
tablero[5][5] = True
tablero[6][5] = True

for y in range(filas):
    for x in range(columnas):
        if tablero[y][x]:
            print('*', end='')
        else:
            print('.', end='')
    print()
```

Aquí tienes lo que muestra por pantalla, de momento, el programa:

```
.........
.........
.........
.........
.....*....
.....*....
.....*....
.........
.........
.........
```

Sigamos. El mundo del juego está gobernado por un reloj. Nosotros seguiremos la evolución del juego durante un número determinado de pulsos. Fijemos, de momento, el número de pulsos a 6:

```
vida.py
pulsos = 6
for t in range(pulsos):
    Acciones asociadas a cada pulso de reloj
```

¿Qué acciones asociamos a cada pulso? Primero, actualizar el tablero, y segundo, mostrarlo:

```
vida.py
for t in range(pulsos):
    Actualizar el tablero

    # Representar el tablero.
    print('Pulso', t+1)
    for y in range(filas):
        for x in range(columnas):
            if tablero[y][x]:
                print('*', end='')
            else:
                print('.', end='')
        print()
```

Vamos a actualizar el tablero. Detallemos un poco más esa tarea:

```
vida.py
for t in range(pulsos):
    # Actualizar el tablero.
    for y in range(filas):
        for x in range(columnas):
            # Calcular el número de vecinos de la celda que estamos visitando.
            n = calcular el número de vecinos
            # Aplicar las reglas.
            if tablero[y][x] and (n == 2 or n == 3): # Supervivencia
                tablero[y][x] = True
            elif not tablero[y][x] and n == 3: # Nacimiento
                tablero[y][x] = True
```

```
            else: # Superpoblación y aislamiento
                tablero[y][x] = False

    # Representar el tablero.
    ...
```

Solo nos falta determinar el número de vecinos. ¿Cómo lo hacemos? Fácil: consultando cada una de las casillas vecinas e incrementando un contador (inicializado a cero) cada vez que encontremos una célula viva:

vida.py

```
filas = 10
columnas = 10

tablero = []
for i in range(filas):
    tablero.append([False]*columnas)

tablero[4][5] = True
tablero[5][5] = True
tablero[6][5] = True

# Representar el tablero.
print('Estado␣inicial')
for y in range(filas):
    for x in range(columnas):
        if tablero[y][x]:
            print('*', end='')
        else:
            print('.', end='')
    print()

pulsos = 6
for t in range(pulsos):
    # Actualizar el tablero.
    for y in range(filas):
        for x in range(columnas):
            # Calcular el número de vecinos de la celda que estamos visitando.
            n = 0
            if tablero[y-1][x-1]:
                n += 1
            if tablero[ y ][x-1]:
                n += 1
            if tablero[y+1][x-1]:
                n += 1
            if tablero[y-1][ x ]:
                n += 1
            if tablero[y+1][ x ]:
                n += 1
            if tablero[y-1][x+1]:
                n += 1
            if tablero[ y ][x+1]:
                n += 1
            if tablero[y+1][x+1]:
                n += 1
            # Aplicar las reglas.
            if tablero[y][x] and (n == 2 or n == 3): # Supervivencia
                tablero[y][x] = True
            elif not tablero[y][x] and n == 3: # Nacimiento
                tablero[y][x] = True
            else: # Superpoblación y aislamiento
```

```
                tablero[y][x] = False
    # Representar el tablero.
    print('Pulso', t+1)
    for y in range(filas):
        for x in range(columnas):
            if tablero[y][x]:
                print('*', end='')
            else:
                print('.', end='')
        print()
```

Ya está. Ejecutemos el programa:

```
Estado inicial
..........
..........
..........
..........
.....*....
.....*....
.....*....
..........
..........
..........
Traceback (most recent call last):
  File "vida.py", line 39, in <module>
    if tablero[y-1][x+1]:
IndexError: list index out of range
```

¿Qué ha ido mal? Python nos dice que nos hemos salido de rango al acceder a un elemento de la matriz. Ya está claro: cuando *x* vale *columnas*-1, *x*+1 vale *columnas* y nos salimos del rango válido de índices. (Hay un problema similar cuando *x* vale 0 y tratamos de consultar la columna *x*-1, solo que no se produce un error de ejecución porque la columna de índice -1 existe: ¡es la columna *columnas*-1!). El juego de la vida original asume que el tablero es infinito. Nosotros hemos de jugar con un tablero que tiene límites, así que tendremos que tratar de modo especial las casillas fronterizas, pues no tienen 8 casillas colindantes. Esta nueva versión tiene esa precaución:

vida.py

```
filas = 10
columnas = 10

tablero = []
for i in range(filas):
    tablero.append([False]*columnas)

tablero[4][5] = True
tablero[5][5] = True
tablero[6][5] = True

# Representar el tablero.
print('Estado inicial')
for y in range(filas):
    for x in range(columnas):
        if tablero[y][x]:
            print('*', end='')
        else:
            print('.', end='')
    print()

pulsos = 6
for t in range(pulsos):
    # Actualizar el tablero.
    for y in range(filas):
```

```
            for x in range(columnas):
                # Calcular el número de vecinos de la celda que estamos visitando.
                n = 0
                if y > 0 and x > 0 and tablero[y-1][x-1]:
                    n += 1
                if x > 0 and tablero[ y ][x-1]:
                    n += 1
                if y < filas-1 and x > 0 and tablero[y+1][x-1]:
                    n += 1
                if y > 0 and tablero[y-1][ x ]:
                    n += 1
                if y < filas-1 and tablero[y+1][ x ]:
                    n += 1
                if y > 0 and x < columnas-1 and tablero[y-1][x+1]:
                    n += 1
                if x < columnas-1 and tablero[ y ][x+1]:
                    n += 1
                if y < filas-1 and x < columnas-1 and tablero[y+1][x+1]:
                    n += 1
                # Aplicar las reglas.
                if tablero[y][x] and (n == 2 or n == 3): # Supervivencia
                    tablero[y][x] = True
                elif not tablero[y][x] and n == 3: # Nacimiento
                    tablero[y][x] = True
                else: # Superpoblación y aislamiento
                    tablero[y][x] = False
        # Representar el tablero.
        print('Pulso', t+1)
        for y in range(filas):
            for x in range(columnas):
                if tablero[y][x]:
                    print('*', end='')
                else:
                    print('.', end='')
            print()
```

Ejecutemos ahora el programa:

```
Estado inicial
..........
..........
..........
..........
.....*....
.....*....
.....*....
..........
..........
..........
Pulso 1
..........
..........
..........
..........
..........
..........
..........
..........
..........
..........
Pulso 2
..........
..........
..........
..........
```

```
..........
..........
..........
..........
..........
..........
Pulso 3
..........
..........
..........
..........
..........
..........
..........
..........
..........
..........
Pulso 4
..........
..........
..........
..........
..........
..........
..........
..........
..........
..........
Pulso 5
..........
..........
..........
..........
..........
..........
..........
..........
..........
..........
Pulso 6
..........
..........
..........
..........
..........
..........
..........
..........
..........
..........
```

¡Alto! ¿Qué ha pasado? ¡No aparece el patrón de oscilación que esperábamos! Haz una traza para ver si averiguas qué ha pasado. Date un poco de tiempo antes de seguir leyendo.

De acuerdo. Confiamos en que has reflexionado un poco y ya has encontrado una explicación de lo ocurrido antes de leer esto. Confirma que estás en lo cierto: ha ocurrido que estamos aplicando las reglas sobre un tablero que se modifica *durante la propia aplicación de las reglas*, y eso no es válido. Numeremos algunas celdas afectadas por el oscilador para explicar lo ocurrido:

```
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . 1 . . . .
. . . . 2 3 4 . . .
. . . . . 5 . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
```

Cuando hemos procesado la celda 1, su número de vecinos era 0 así que ha muerto (regla de aislamiento). La celda 2 pasa entonces a tener 2 vecinos, así que muere. Si la celda 1 no hubiera muerto aún, hubiésemos contado 3 vecinos, y la celda 2 hubiese pasado a estar viva (regla de nacimiento). La celda 3 tiene ahora 1 vecino, luego muere (lo correcto hubiera sido contar 2 vecinos y aplicar la regla de supervivencia). La celda 4 cuenta con un solo vecino (deberían haber sido 3), luego muere. Y la celda 5 no tiene vecinos, luego también muere. Resultado: todas las células mueren.

¿Cómo podemos ingeniar un método que no mate/resucite células durante el propio pulso? Una técnica sencilla consiste en usar dos tableros. Uno de ellos no se modifica durante la aplicación de las reglas y los vecinos se cuentan sobre su configuración. La nueva configuración se va calculando y escribiendo en el segundo tablero. Cuando finaliza el proceso, el tablero actual copia su contenido del tablero nuevo. Te ofrecemos ya una versión completa del juego:

vida.py
```
filas = 10
columnas = 10

tablero = []
for i in range(filas):
    tablero.append([False]*columnas)

tablero[4][5] = True
tablero[5][5] = True
tablero[6][5] = True

# Representar el tablero.
print('Estado inicial')
for y in range(filas):
    for x in range(columnas):
        if tablero[y][x]:
            print('*', end='')
        else:
            print('.', end='')
    print()

pulsos = 6
for t in range(pulsos):
    # Preparar un nuevo tablero.
    nuevo = []
    for i in range(filas):
        nuevo.append([False]*columnas)

    # Actualizar el tablero.
    for y in range(filas):
        for x in range(columnas):
            # Calcular el número de vecinos de la celda que estamos visitando.
            n = 0
            if y > 0 and x > 0 and tablero[y-1][x-1]:
                n += 1
            if x > 0 and tablero[ y ][x-1]:
                n += 1
            if y < filas-1 and x > 0 and tablero[y+1][x-1]:
                n += 1
            if y > 0 and tablero[y-1][ x ]:
                n += 1
            if y < filas-1 and tablero[y+1][ x ]:
                n += 1
            if y > 0 and x < columnas-1 and tablero[y-1][x+1]:
                n += 1
            if x < columnas-1 and tablero[ y ][x+1]:
                n += 1
```

```
            if y < filas-1 and x < columnas-1 and tablero[y+1][x+1]:
                n += 1
            # Aplicar las reglas.
            if tablero[y][x] and (n == 2 or n == 3): # Supervivencia
                nuevo[y][x] = True
            elif not tablero[y][x] and n == 3: # Nacimiento
                nuevo[y][x] = True
            else: # Superpoblación y aislamiento
                nuevo[y][x] = False

    # Actualizar el tablero.
    tablero = nuevo

    # Representar el tablero.
    print('Pulso', t+1)
    for y in range(filas):
        for x in range(columnas):
            if tablero[y][x]:
                print('*', end='')
            else:
                print('.', end='')
        print()
```

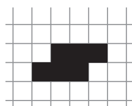
```
Estado inicial
..........
..........
..........
..........
.....*....
.....*....
.....*....
..........
..........
..........
Pulso 1
..........
..........
..........
..........
..........
....***...
..........
..........
..........
..........
Pulso 2
..........
..........
..........
..........
.....*....
.....*....
.....*....
..........
..........
..........
Pulso 3
..........
..........
..........
..........
..........
....***...
..........
..........
..........
```

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

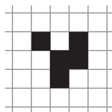
```
..........
Pulso 4
..........
..........
..........
..........
.....*....
.....*....
.....*....
..........
..........
..........
Pulso 5
..........
..........
..........
..........
..........
....***...
..........
..........
..........
..........
Pulso 6
..........
..........
..........
..........
.....*....
.....*....
.....*....
..........
..........
..........
```

Ahora sí. Puedes probar algunas configuraciones del juego de la vida tan interesantes que tienen nombre propio (conviene que los pruebes en tableros de gran dimensión):

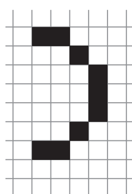
- La rana:



- El deslizador:



- La abeja reina lanzadora:



▶ 248 ¿Funciona esta otra forma de contar los vecinos de la casilla de la fila *y* y columna *x*?

```
if tablero[y][x]:
    n = -1
```

¿El juego del universo?

El juego de la vida fue inventado en 1970 por el matemático John H. Conway y popularizado por Martin Gardner en su columna de Scientific American. El juego de la vida es un caso particular de autómata celular, un sistema en el que ciertas reglas deciden acerca del valor que debe tomar una celda en un tablero a partir de los valores de sus vecinas.

Los autómatas celulares ilustran la denominada «complejidad emergente», un campo relativamente reciente dedicado a estudiar la aparición de patrones complejos y la autoorganización a partir de reglas simples. Parecen proporcionar un buen modelo para numerosos fenómenos naturales, como la pigmentación en conchas y otros animales.

Una hipótesis interesante es que la naturaleza no es más que un superordenador que está jugando alguna variante del juego de la vida. ¿Una idea extravagante? Stephen Wolfram, el autor principal del celebrado programa *Mathematica*, se ha encerrado una década para investigar esta cuestión. El resultado: un polémico libro titulado «A new kind of science» en el que propone «un nuevo tipo de ciencia» para estudiar el funcionamiento del universo a partir del análisis y observación de autómatas celulares.

Internet está plagada de páginas web dedicadas al juego de la vida y a los autómatas celulares. Búscalas y diviértete con la infinidad de curiosos patrones que generan las formas más increíbles.

```
else:
    n = 0
for i in [-1, 0, 1]:
    for j in [-1, 0, 1]:
        if y+i >= 0 and y+i < filas and x+j >= 0 and x+j < columnas:
            if tablero[y+i][x+j]:
                n += 1
```

▶ 249 El «juego de la vida parametrizado» es una generalización del juego de la vida. En él, el número de vecinos vivos necesarios para activar las reglas de nacimiento, supervivencia, aislamiento y superpoblación están parametrizados. Haz un programa que solicite al usuario el número de células vecinas vivas necesarias para que se disparen las diferentes reglas y muestre cómo evoluciona el tablero con ellas.

▶ 250 El juego de la vida toroidal se juega sobre un tablero de dimensión finita $m \times n$ con unas reglas de vecindad diferentes. Una casilla de coordenadas (y, x) tiene siempre 8 vecinas, aunque esté en un borde:

$((y-1) \text{ mód } m, (x-1) \text{ mód } n)$	$((y-1) \text{ mód } m, x)$	$((y-1) \text{ mód } m, (x+1) \text{ mód } n)$
$(y, (x-1) \text{ mód } n)$		$(y, (x+1) \text{ mód } n)$
$((y+1) \text{ mód } m, (x-1) \text{ mód } n)$	$((y+1) \text{ mód } m, x)$	$((y+1) \text{ mód } m, (x+1) \text{ mód } n)$

donde mód es el operador módulo (en Python, `%`).

Implementa el juego de la vida toroidal con los gráficos de tortuga.

▶ 251 El juego de la vida es un tipo particular de *autómata celular bidimensional*. Hay autómatas celulares unidimensionales. En ellos, una lista de valores (en su versión más simple, ceros y unos) evoluciona a lo largo del tiempo a partir del estado de sus celdas vecinas (solo las celdas izquierda y derecha en su versión más simple) y de ella misma en el instante anterior.

Por ejemplo, una regla $001 \rightarrow 1$ se lee como «la célula está viva si en la iteración anterior estaba muerta y tenía una célula muerta a la izquierda y una célula viva a la derecha». Una especificación completa tiene este aspecto:

$$000 \rightarrow 0, 001 \rightarrow 1, 010 \rightarrow 1, 011 \rightarrow 0, 100 \rightarrow 1, 101 \rightarrow 1, 110 \rightarrow 0, 111 \rightarrow 0,$$

Y aquí tienes una representación (usando asteriscos para los unos y puntos para los ceros) de la evolución del sistema durante sus primeros pulsos partiendo de una configuración muy sencilla (un solo uno):

```
Pulso 0 : . . . . . . . . . . . . . * . . . . . . . . . .
Pulso 1 : . . . . . . . . . . . . * * * . . . . . . . . .
Pulso 2 : . . . . . . . . . . . * . . . * . . . . . . . .
Pulso 3 : . . . . . . . . . . * * * . * * * . . . . . . .
Pulso 4 : . . . . . . . . . * . . . * . . . * . . . . . .
Pulso 5 : . . . . . . . . * * * . * * * . * * * . . . . .
Pulso 6 : . . . . . . . * . . . * . . . * . . . * . . . .
```

Implementa un programa para estudiar la evolución de autómatas celulares unidimensionales. El programa leerá un conjunto de reglas por teclado y un número de pulsos. A continuación, mostrará en el terminal de texto la evolución del autómata partiendo de una configuración con solo una celda viva que ocupa la posición central del universo.

Cuando tengas el programa, explora las siguientes reglas:

- 000 → 0, 001 → 1, 010 → 1, 011 → 1, 100 → 1, 101 → 0, 110 → 0, 111 → 0.
- 000 → 0, 001 → 0, 010 → 1, 011 → 1, 100 → 1, 101 → 0, 110 → 0, 111 → 0.
- 000 → 0, 001 → 1, 010 → 1, 011 → 1, 100 → 0, 101 → 1, 110 → 1, 111 → 0.
- 000 → 0, 001 → 1, 010 → 1, 011 → 1, 100 → 0, 101 → 1, 110 → 1, 111 → 0.
- 000 → 0, 001 → 1, 010 → 1, 011 → 0, 100 → 1, 101 → 1, 110 → 0, 111 → 1.

Capítulo 6

Funciones

—Y ellos, naturalmente, responden a sus nombres, ¿no? —observó al desgaire el Mosquito.

—Nunca oí decir tal cosa.

—¿Pues de qué les sirve tenerlos —preguntó el Mosquito— si no responden a sus nombres?

Alicia en el país de las maravillas, Lewis Carroll

En capítulos anteriores hemos aprendido a utilizar funciones. Algunas de ellas están predefinidas (*abs*, *round*, etc.) mientras que otras deben importarse de módulos antes de poder ser usadas (por ejemplo, *sin* y *cos* se importan del módulo *math*). En este tema aprenderemos a definir nuestras propias funciones. Definiendo nuevas funciones estaremos «enseñando» a Python a hacer cálculos que inicialmente no sabe hacer y, en cierto modo, adaptando el lenguaje de programación al tipo de problemas que deseamos resolver, enriqueciéndolo para que el programador pueda ejecutar acciones complejas de un modo sencillo: llamando a funciones desde su programa.

Ya has usado módulos, es decir, ficheros que contienen funciones y variables de valor predefinido que puedes importar en tus programas. En este capítulo aprenderemos a crear nuestros propios módulos, de manera que reutilizar nuestras funciones en varios programas resultará extremadamente sencillo: bastará con importarlas.

6.1. Uso de funciones

Denominaremos *activar*, *invocar* o *llamar* a una función a la acción de usarla. Las funciones que hemos aprendido a invocar reciben cero, uno o más *argumentos* separados por comas y encerrados entre un par de paréntesis y pueden devolver un valor o no devolver nada.

```
>>> abs(-3)↵
3
>>> abs(round(2.45, 1))↵
2.5
>>> from math import sin↵
>>> sin(1)↵
0.8414709848078965
```

Podemos llamar a una función desde una expresión. Como el resultado tiene un tipo determinado, hemos de estar atentos a que este sea compatible con la operación y tipo de los operandos con los que se combina:

```
>>> 1 + (abs(-3) * 2)↵
7
>>> 2.5 / abs(round(2.45, 1))↵
1.0
>>> 3 + str(3)↵
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

¿Ves? En el último caso se ha producido un error de tipos porque se ha intentado sumar una cadena, que es el tipo de dato del valor devuelto por *str*, a un entero.

Observa que los argumentos de una función también pueden ser expresiones:

```
>>> abs(round(1.0/9, 4//(1+1)))↵
0.11
```

6.2. Definición de funciones

Vamos a estudiar el modo en que podemos definir (y usar) nuestras propias funciones Python. Estudiaremos en primer lugar cómo definir y llamar a funciones que devuelven un valor y pasaremos después a presentar los denominados procedimientos: funciones que no devuelven ningún valor. Además de los conceptos y técnicas que te iremos presentando, es interesante que te fijes en cómo desarrollamos los diferentes programas de ejemplo.

6.2.1. Definición y uso de funciones con un solo parámetro

Empezaremos definiendo una función muy sencilla, una que recibe un número y devuelve el cuadrado de dicho número. El nombre que daremos a la función es *cuadrado*. Observa este fragmento de programa:

```
cuadrado.py
def cuadrado(x):
    return x ** 2
```

Ya está. Acabamos de definir la función *cuadrado* que se aplica sobre un valor al que llamamos *x* y devuelve un número: el resultado de elevar *x* al cuadrado. En el programa aparecen dos nuevas palabras reservadas: **def** y **return**. La palabra **def** es abreviatura de «define» y **return** significa «devuelve» en inglés. Podríamos leer el programa anterior como «define cuadrado de *x* como el valor que resulta de elevar *x* al cuadrado».

En las líneas que siguen a su definición, la función *cuadrado* puede utilizarse del mismo modo que las funciones predefinidas:

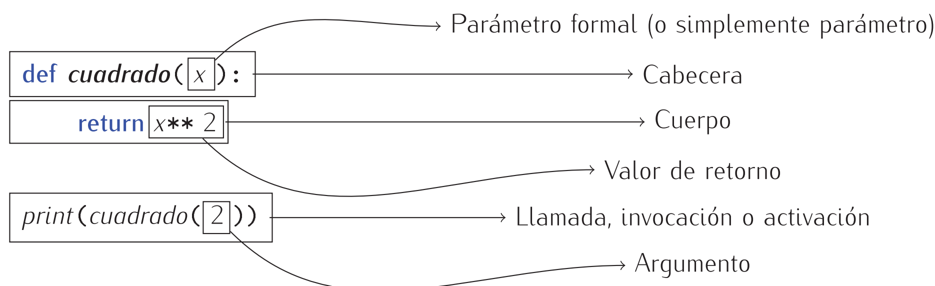
```
cuadrado.py
1 def cuadrado(x):
2     return x ** 2
3
4 print(cuadrado(2))
5 a = 1 + cuadrado(3)
6 print(cuadrado(a * 3))
```

Este es el resultado de ejecutar el programa:

```
4
900
```

En cada caso, el resultado de la expresión que sigue entre paréntesis al nombre de la función es utilizado como valor de *x* durante la ejecución de *cuadrado*. En la primera llamada (línea 4) el valor es 2, en la siguiente llamada es 3 y en la última, 30. Fácil, ¿no?

Detengámonos un momento para aprender algunos términos nuevos. La línea que empieza con **def** es la *cabecera* de la función y el fragmento de programa que contiene los cálculos que debe efectuar la función se denomina *cuerpo* de la función. Cuando estamos definiendo una función, su parámetro se denomina *parámetro formal* (aunque, por abreviar, normalmente usaremos el término *parámetro*, sin más). El valor que *pasamos* a una función cuando la invocamos se denomina *parámetro real* o *argumento*. Las porciones de un programa que no son cuerpo de funciones forman parte del *programa principal*: son las sentencias que se ejecutarán cuando el programa entre en acción. El cuerpo de las funciones solo se ejecutará si se producen las correspondientes llamadas.



Definir no es invocar

Si intentamos ejecutar este programa:

cuadrado.py
```
def cuadrado(x):
    return x ** 2
```

no ocurrirá nada en absoluto; bueno, al menos nada que aparezca por pantalla. La definición de una función solo hace que Python «aprenda» *silenciosamente* un método de cálculo asociado al identificador *cuadrado*. Nada más. Hagamos la prueba ejecutando el programa:

```
python cuadrado.py
```

¿Lo ves? No se ha impreso nada en pantalla. No se trata de que no haya ningún *print*, sino de que definir una función es un proceso que no tiene eco en pantalla. Repetimos: definir una función solo asocia un método de cálculo a un identificador y no supone ejecutar dicho método de cálculo.

Este otro programa sí muestra algo por pantalla:

cuadrado.py
```
1 def cuadrado(x):
2     return x ** 2
3
4 print(cuadrado(2))
```

Al invocar a la función *cuadrado* (línea 4) se ejecuta esta. En el programa, la invocación de la última línea provoca la ejecución de la línea 2 con un valor de *x* igual a 2 (argumento de la llamada). El valor devuelto con **return** es mostrado en pantalla como efecto de la sentencia *print* de la línea 4. Hagamos la prueba:

```
python cuadrado.py
4
```

Las reglas para dar nombre a las funciones y a sus parámetros son las mismas que seguimos para dar nombre a las variables: solo se pueden usar letras, dígitos y el carácter de subrayado; la primera letra del nombre no puede ser un número; y no se pueden usar palabras reservadas. Pero, ¡cuidado!: no debes dar el mismo nombre a una función y a una variable. En Python, cada nombre debe identificar claramente un único elemento: una variable o una función[1].

Al definir una función *cuadrado* es como si hubiésemos creado una «máquina de calcular cuadrados». Desde la óptica de su uso, podemos representar la función como una caja que transforma un *dato de entrada* en un *dato de salida*:

[1] Más adelante, al presentar las variables locales, matizaremos esta afirmación.

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

Definición de funciones desde el entorno interactivo

Hemos aprendido a definir funciones dentro de un programa. También puedes definir funciones desde el entorno interactivo de Python. Te vamos a enseñar paso a paso qué ocurre en el entorno interactivo cuando estamos definiendo una función.

En primer lugar aparece el *prompt*. Podemos escribir entonces la primera línea:

```
>>> def cuadrado(x):↵
...↵
```

Python nos responde con tres puntos (`...`). Esos tres puntos son el llamado *prompt secundario*: indica que la acción de definir la función no se ha completado aún y nos pide más sentencias. Escribimos a continuación la segunda línea respetando el sangrado que le corresponde:

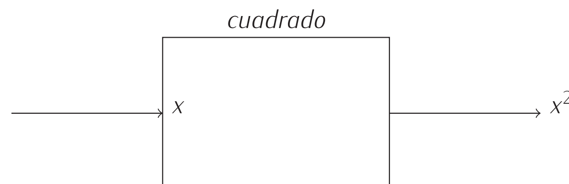
```
>>> def cuadrado(x):↵
...     return x ** 2↵
```

Nuevamente Python responde con el *prompt* secundario. Es necesario que le demos una vez más al retorno de carro para que Python entienda que ya hemos acabado de definir la función:

```
>>> def cuadrado(x):↵
...     return x ** 2↵
...↵
>>>↵
```

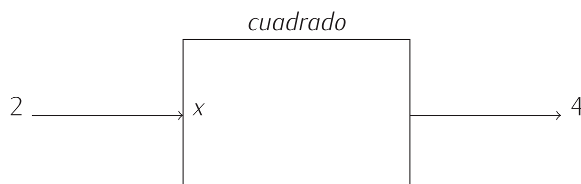
Ahora aparece de nuevo el *prompt* principal o primario. Python ha aprendido la función y está listo para que introduzcamos nuevas sentencias o expresiones.

```
>>> def cuadrado(x):↵
...     return x ** 2↵
... ↵
>>> cuadrado(2)↵
4
>>> 1 + cuadrado(1+3)↵
17
```



Cuando invocas a la función, le estás «conectando» un valor a la entrada, así que la «máquina de calcular cuadrados» se pone en marcha y produce la solución deseada:

```
>>> cuadrado(2)↵
4
```



Ojo: no hay una única forma de construir la «máquina de calcular cuadrados». Fíjate en esta definición alternativa:

```
cuadrado.py
def cuadrado(x):
    return x * x
```

Se trata de una definición tan válida como la anterior, ni mejor, ni peor. Como usuarios de la función, poco nos importa *cómo* hace el cálculo[2]; lo que importa es *qué datos recibe* y *qué valor devuelve*.

Vamos con un ejemplo más: una función que calcula el valor de x por el seno de x:

```
from math import sin

def xsin(x):
    return x * sin(x)
```

Lo interesante de este ejemplo es que la función definida, *xsin*, contiene una llamada a otra función (*sin*). No hay problema: desde una función puedes invocar a cualquier otra.

Una confusión frecuente

Supongamos que definimos una función con un parámetro x como esta:

```
def cubo(x):
    return x ** 3
```

Es frecuente en los aprendices confundir el parámetro x con una variable x. Así, les parece extraño que podamos invocar a la función de este modo:

```
y = 1
print(cubo(y))
```

¿Cómo es que ahora llamamos y a lo que se llamaba x? No hay problema alguno. Al definir una función, usamos un identificador cualquiera para referirnos al parámetro. Tanto da que se llame x como y. Esta otra definición de cubo es absolutamente equivalente:

```
def cubo(z):
    return z ** 3
```

La definición se puede leer así: «si te pasan un valor, digamos z, devuelve ese valor elevado al cubo». Usamos el nombre z (o x) solo para poder referirnos a él en el cuerpo de la función.

▶ 252 Define una función llamada *raíz_cúbica* que devuelva el valor de $\sqrt[3]{x}$.

(Nota: recuerda que la notación $\sqrt[3]{x}$ no es más que una forma de expresar $x^{1/3}$).

▶ 253 Define una función llamada *área_círculo* que, a partir del radio de un círculo, devuelva el valor de su área. Utiliza el valor 3.1416 como aproximación de π o importa el valor de π que encontrarás en el módulo *math*.

(Recuerda que el área de un círculo de radio r es πr^2).

▶ 254 Define una función que convierta grados Farenheit en grados centígrados.

(Para calcular los grados centígrados has de restar 32 a los grados Farenheit y multiplicar el resultado por cinco novenos).

▶ 255 Define una función que convierta grados centígrados en grados Farenheit.

▶ 256 Define una función que convierta radianes en grados.

(Recuerda que 360 grados son 2π radianes).

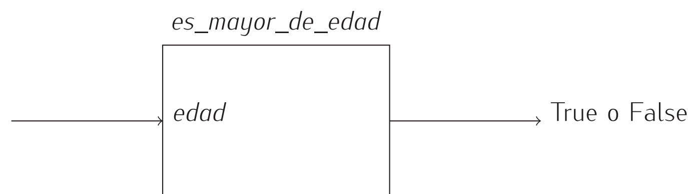
▶ 257 Define una función que convierta grados en radianes.

---

[2]. . . por el momento. Hay muchas formas de hacer el cálculo, pero unas resultan más *eficientes* (más rápidas) que otras. Naturalmente, cuando podamos elegir, escogeremos la forma más eficiente.

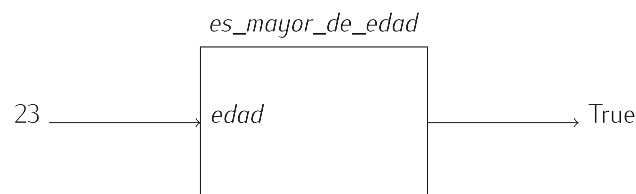
Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

En el cuerpo de una función no solo pueden aparecer sentencias return; también podemos usar estructuras de control: sentencias condicionales, bucles, etc. Lo podemos comprobar diseñando una función que recibe un número y devuelve un booleano. El valor de entrada es la edad de una persona y la función devuelve True si la persona es mayor de edad y False en caso contrario:

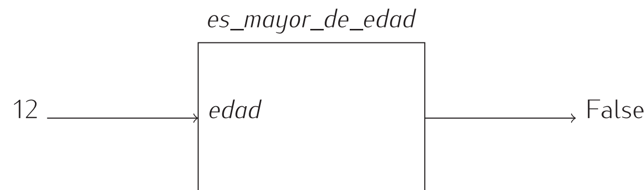


Cuando llamas a la función, esta se activa para producir un resultado concreto (en nuestro caso, o bien devuelve True o bien devuelve False):

```
a = es_mayor_de_edad(23)
```




```
b = es_mayor_de_edad(12)
```



Una forma usual de devolver valores de función es a través de un solo return ubicado al final del cuerpo de la función:

```
mayoria_edad.py
def es_mayor_de_edad(edad):
    if edad < 18:
        resultado = False
    else:
        resultado = True
    return resultado
```

Pero no es el único modo en que puedes devolver diferentes valores. Mira esta otra definición de la misma función:

```
mayoria_edad.py
def es_mayor_de_edad(edad):
    if edad < 18:
        return False
    else:
        return True
```

Aparecen dos sentencias return: cuando la ejecución llega a cualquiera de ellas, finaliza *inmediatamente* la llamada a la función y se devuelve el valor que sigue al return. Podemos

asimilar el comportamiento de **return** al de **break**: una sentencia **break** fuerza a terminar la ejecución de un bucle y una sentencia **return** fuerza a terminar la ejecución de una llamada a función.

▶ 258 ¿Es este programa equivalente al que acabamos de ver?

mayoria_edad.py
```
def es_mayor_de_edad(edad):
    if edad < 18:
        return False
    return True
```

▶ 259 ¿Es este programa equivalente al que acabamos de ver?

mayoria_edad.py
```
def es_mayor_de_edad(edad):
    return edad >= 18
```

▶ 260 La última letra del DNI puede calcularse a partir del número. Para ello solo tienes que dividir el número por 23 y quedarte con el resto, que es un número entre 0 y 22. La letra que corresponde a cada número la tienes en esta tabla:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
T	R	W	A	G	M	Y	F	P	D	X	B	N	J	Z	S	Q	V	H	L	C	K	E

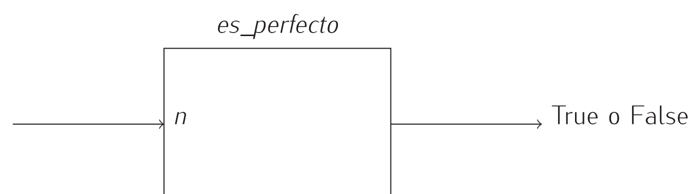
Define una función llamada *letra_dni* que, dado un número de DNI, devuelva la letra que le corresponde.

▶ 261 Diseña una función que reciba una cadena y devuelva cierto si empieza por minúscula y falso en caso contrario.

▶ 262 Diseña una función llamada *es_repetición* que reciba una cadena y nos diga si la cadena está formada mediante la concatenación de una cadena consigo misma. Por ejemplo, *es_repetición*('abab') devolverá True, pues la cadena 'abab' está formada con la cadena 'ab' repetida; por contra *es_repetición*('ababab') devolverá False.

Y ahora, un problema más complicado. Vamos a diseñar una función que nos diga si un número dado es o no es *perfecto*. Se dice que un número es perfecto si es igual a la suma de todos sus divisores excluído él mismo. Por ejemplo, 28 es un número perfecto, pues sus divisores (excepto él mismo) son 1, 2, 4, 7 y 14, que suman 28.

Empecemos. La función, a la que llamaremos *es_perfecto*, recibirá un solo dato (el número sobre el que hacemos la pregunta) y devolverá un valor booleano:



La cabecera de la función está clara:

perfecto.py
```
def es_perfecto(n):
    ...
```

¿Y por dónde seguimos? Vamos por partes. En primer lugar estamos interesados en conocer todos los divisores del número. Una vez tengamos claro cómo saber cuáles son, los sumaremos. Si la suma coincide con el número original, este es perfecto; si no, no. Podemos usar un bucle y preguntar a todos los números entre 1 y *n*-1 si son divisores de *n*:

```
perfecto.py
def es_perfecto(n):
    for i in range(1, n):
        if i es divisor de n:
            ...
```

Observa cómo seguimos siempre las reglas de sangrado de código que impone Python. ¿Y cómo preguntamos ahora si un número es divisor de otro? El operador módulo % devuelve el resto de la división y resuelve fácilmente la cuestión:

```
perfecto.py
1 def es_perfecto(n):
2     for i in range(1, n):
3         if n % i == 0:
4             ...
```

La línea 4 solo se ejecutará para valores de *i* que son divisores de *n*. ¿Qué hemos de hacer a continuación? Deseamos sumar todos los divisores y ya conocemos la «plantilla» para calcular sumatorios:

```
perfecto.py
def es_perfecto(n):
    sumatorio = 0
    for i in range(1, n):
        if n % i == 0:
            sumatorio += i
    ...
```

¿Qué queda por hacer? Comprobar si el número es perfecto y devolver True o False, según proceda:

```
perfecto.py
def es_perfecto(n):
    sumatorio = 0
    for i in range(1, n):
        if n % i == 0:
            sumatorio += i
    if sumatorio == n:
        return True
    else:
        return False
```

Y ya está. Bueno, podemos simplificar un poco las cuatro últimas líneas y convertirlas en una sola. Observa esta nueva versión:

```
perfecto.py
def es_perfecto(n):
    sumatorio = 0
    for i in range(1, n):
        if n % i == 0:
            sumatorio += i
    return sumatorio == n
```

¿Qué hace la última línea? Devuelve el resultado de evaluar la expresión lógica que compara *sumatorio* con *n*: si ambos números son iguales, devuelve True, y si no, devuelve False. Mejor, ¿no?

▶ 263 ¿En qué se ha equivocado nuestro aprendiz de programador al escribir esta función?

perfecto.py
```
def es_perfecto(n):
    for i in range(1, n):
        sumatorio = 0
        if n % i == 0:
            sumatorio += i
    return sumatorio == n
```

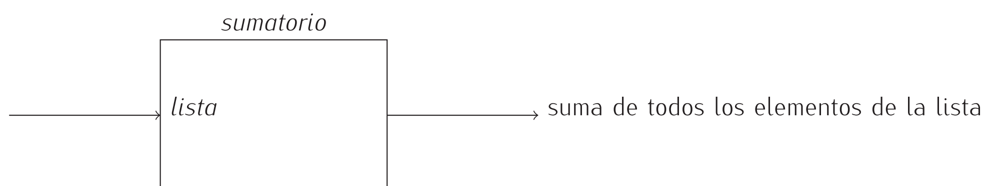
▶ 264 Mejora la función *es_perfecto* haciéndola más rápida. ¿Es realmente necesario considerar todos los números entre 1 y *n*-1?

▶ 265 Diseña una función que devuelva una lista con los números perfectos comprendidos entre 1 y *n*, siendo *n* un entero que nos proporciona el usuario.

▶ 266 Define una función que devuelva el número de días que tiene un año determinado. Ten en cuenta que un año es bisiesto si es divisible por 4 y no divisible por 100, excepto si es también divisible por 400, en cuyo caso es bisiesto.

(Ejemplos: El número de días de 2002 es 365: el número 2002 no es divisible por 4, así que no es bisiesto. El año 2004 es bisiesto y tiene 366 días: el número 2004 es divisible por 4, pero no por 100, así que es bisiesto. El año 1900 es divisible por 4, pero no es bisiesto porque es divisible por 100 y no por 400. El año 2000 sí es bisiesto: el número 2000 es divisible por 4 y, aunque es divisible por 100, también lo es por 400).

Hasta el momento nos hemos limitado a suministrar valores escalares como argumentos de una función, pero también es posible suministrar argumentos de tipo secuencial. Veámoslo con un ejemplo: una función que recibe una lista de números y nos devuelve el sumatorio de todos sus elementos.



suma_lista.py
```
def sumatorio(lista):
    suma = 0
    for número in lista:
        suma += número
    return suma
```

Podemos usar la función así:

suma_lista.py
```
def sumatorio(lista):
    suma = 0
    for número in lista:
        suma += número
    return suma

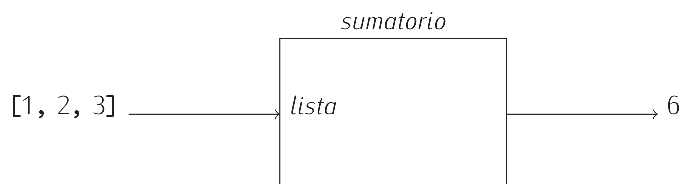
a = [1, 2, 3]
print(sumatorio(a))
```

o así:

```
suma_lista.py
def sumatorio(lista):
    suma = 0
    for número in lista:
        suma += número
    return suma

print(sumatorio([1, 2, 3]))
```

En cualquiera de los dos casos, el parámetro *lista* toma el valor [1, 2, 3], que es el argumento suministrado en la llamada:



Sumatorios

Has aprendido a calcular sumatorios con bucles. Desde la versión 2.3, Python ofrece una forma mucho más cómoda de calcular sumatorios: la función predefinida *sum*, que recibe una lista de valores y devuelve el resultado de sumarlos.

```
>>> sum([1, 10, 20])↵
31
```

La función *sum* (y también la que hemos diseñado, *sumatorio*), no solo suma elementos de listas: también suma elementos de una sucesión cualquiera. ¿Cómo usarla para calcular el sumatorio de los 100 primeros números naturales? Muy fácil: pasándole una secuencia con esos números, algo que resulta trivial si usas *range*:

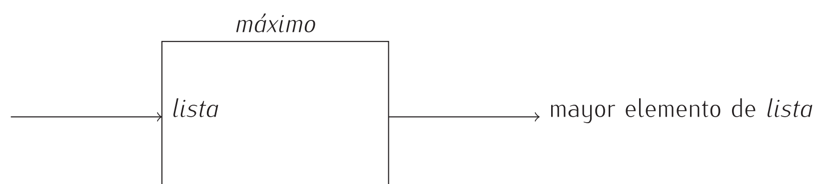
```
>>> sum(range(1, 101))↵
5050
```

Ten cuidado. No es la forma más eficiente de sumar los 100 primeros números. Recuerda que la suma de los n primeros números se puede calcular directamente como $n \cdot (n+1)/2$:

```
>>> def suma_primeros(n):↵
...     return n * (n+1) // 2↵
... ↵
>>> suma_primeros(100)↵
5050
```

▶ 267 Diseña una función que calcule el sumatorio de la diferencia entre números contiguos en una lista. Por ejemplo, para la lista [1, 3, 6, 10] devolverá 9, que es $2+3+4$ (el 2 resulta de calcular $3-1$, el 3 de calcular $6-3$ y el 4 de calcular $10-6$).

Estudiemos otro ejemplo: una función que recibe una lista de números y devuelve el valor de su mayor elemento.



La idea básica es sencilla: recorrer la lista e ir actualizando el valor de una variable auxiliar que, en todo momento, contendrá el máximo valor visto hasta ese momento.

```
maximo.py
def máximo(lista):
    for elemento in lista:
        if elemento > candidato:
            candidato = elemento
    return candidato
```

Nos falta inicializar la variable *candidato*. ¿Con qué valor? Podríamos pensar en inicializarla con el menor valor posible. De ese modo, cualquier valor de la lista será mayor que él y es seguro que su valor se modificará tan pronto empecemos a recorrer la lista. Pero hay un problema: no sabemos cuál es el menor valor posible. Una buena alternativa es inicializar *candidato* con el valor del primer elemento de la lista. Si ya es el máximo, perfecto, y si no lo es, más tarde se modificará *candidato*.

```
maximo.py
1 def máximo(lista):
2     candidato = lista[0]
3     for elemento in lista:
4         if elemento > candidato:
5             candidato = elemento
6     return candidato
```

▶ 268 Haz una traza de la llamada *máximo*([6, 2, 7, 1, 10, 1, 0]).

¿Ya está? Aún no. ¿Qué pasa si se proporciona una lista vacía como entrada? La línea 2 provocará un error de tipo *IndexError*, pues en ella intentamos acceder al primer elemento de la lista... y la lista vacía no tiene ningún elemento. Un objetivo es, pues, evitar ese error. Pero, en cualquier caso, algo hemos de devolver como máximo elemento de una lista, ¿y qué valor podemos devolver como máximo elemento de una lista vacía? Mmmm. A bote pronto, tenemos dos posibilidades:

- Devolver un valor especial, como el valor 0. Mejor no. Tiene un serio inconveniente: ¿cómo distinguiré el máximo de [-3, -5, 0, -4], que es un cero «legítimo», del máximo de []?
- O devolver un valor «muy» especial, como el valor None. ¿Que qué es None? None significa en inglés «ninguno» y es un valor predefinido en Python que se usa para denotar «ausencia de valor». Como el máximo de una lista vacía no existe, parece acertado devolver la «ausencia de valor» como máximo de sus miembros.

Nos inclinamos por esta segunda opción. En adelante, usaremos None siempre que queramos referirnos a un valor «muy» especial: a la ausencia de valor.

```
maximo.py
def máximo(lista):
    if len(lista) > 0:
        candidato = lista[0]
        for elemento in lista:
            if elemento > candidato:
                candidato = elemento
    else:
        candidato = None
    return candidato
```

▶ 269 Diseña una función que, dada una lista de números enteros, devuelva el número de «series» que hay en ella. Llamamos «serie» a todo tramo de la lista con valores idénticos.

Por ejemplo, la lista [1, 1, 8, 8, 8, 8, 0, 0, 0, 2, 10, 10] tiene 5 «series» (ten en cuenta que el 2 forma parte de una «serie» de un solo elemento).

▶ 270 Diseña una función que diga en qué posición empieza la «serie» más larga de una lista. En el ejemplo del ejercicio anterior, la «serie» más larga empieza en la posición 2 (que es el índice donde aparece el primer 8). (Nota: si hay dos «series» de igual longitud y esta es la mayor, debes devolver la posición de la primera de las «series». Por ejemplo, para [8, 2, 2, 9, 9] deberás devolver la posición 1).

▶ 271 Haz una función que reciba una lista de números y devuelva la media de dichos números. Ten cuidado con la lista vacía (su media es cero).

▶ 272 Diseña una función que calcule el productorio de todos los números que componen una lista.

▶ 273 Diseña una función que devuelva el valor absoluto de la máxima diferencia entre dos elementos consecutivos de una lista. Por ejemplo, el valor devuelto para la lista [1, 10, 2, 6, 2, 0] es 9, pues es la diferencia entre el valor 1 y el valor 10.

▶ 274 Diseña una función que devuelva el valor absoluto de la máxima diferencia entre cualquier par de elementos de una lista. Por ejemplo, el valor devuelto para la lista [1, 10, 2, 6, 8, 2, 0] es 10, pues es la diferencia entre el valor 10 y el valor 0. (Pista: te puede convenir conocer el valor máximo y el valor mínimo de la lista).

▶ 275 Define una función que, dada una cadena *x*, devuelva otra cuyo contenido sea el resultado de concatenar 6 veces *x* consigo misma.

▶ 276 Diseña una función que, dada una lista de cadenas, devuelva la cadena más larga. Si dos o más cadenas miden lo mismo y son las más largas, la función devolverá una cualquiera de ellas.

(Ejemplo: dada la lista ['Pepe', 'Juan', 'María', 'Ana'], la función devolverá la cadena 'María').

▶ 277 Diseña una función que, dada una lista de cadenas, devuelva *una lista con todas* las cadenas más largas, es decir, si dos o más cadenas miden lo mismo y son las más largas, la lista las contendrá a todas.

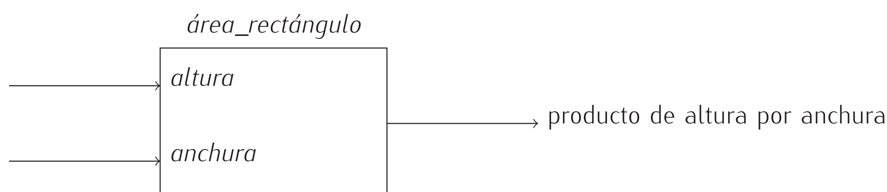
(Ejemplo: dada la lista ['Pepe', 'Ana', 'Juan', 'Paz'], la función devolverá la lista de dos elementos ['Pepe', 'Juan']).

▶ 278 Diseña una función que reciba una lista de cadenas y devuelva el prefijo común más largo. Por ejemplo, la cadena 'pol' es el prefijo común más largo de esta lista:

```
['poliedro', 'policía', 'polífona', 'polinizar', 'polo', 'política']
```

6.2.2. Definición y uso de funciones con varios parámetros

No todas las funciones tienen un solo parámetro. Vamos a definir ahora una con dos parámetros: una función que devuelve el valor del área de un rectángulo dadas su altura y su anchura:



Importaciones, definiciones de función y programa principal

Los programas que diseñes a partir de ahora tendrán tres «tipos de línea»: importación de módulos (o funciones y variables de módulos), definición de funciones y sentencias del programa principal. En principio puedes alternar líneas de los tres tipos. Mira este programa, por ejemplo,

```
def cuadrado(x):
    return x**2

mivector = []
for i in range(3):
    mivector.append(float(input('Dame un número: ')))

def suma_cuadrados(vector):
    suma = 0
    for elemento in vector:
        suma += cuadrado(elemento)
    return suma

s = suma_cuadrados(mivector)

from math import sqrt
print('Distancia al origen:', sqrt(s))
```

En él se alternan definiciones de función, importaciones de funciones y sentencias del programa principal, así que resulta difícil hacerse una idea clara de qué hace el programa. No diseñes así tus programas.

Importaciones, definiciones de función y programa principal (y II)

Esta otra versión del programa anterior pone en primer lugar las importaciones, a continuación, las funciones y, al final, de un tirón, las sentencias que conforman el programa principal:

```
from math import sqrt

def cuadrado(x):
    return x**2

def suma_cuadrados(vector):
    suma = 0
    for elemento in vector:
        suma += cuadrado(elemento)
    return suma

# Programa principal
mivector = []
for i in range(3):
    mivector.append(float(input('Dame un número: ')))
s = suma_cuadrados(mivector)
print('Distancia al origen:', sqrt(s))
```

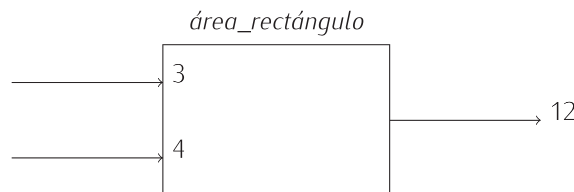
Es mucho más legible. Te recomendamos que sigas siempre esta organización en tus programas. Recuerda que la legibilidad de los programas es uno de los objetivos del programador.

rectangulo.py
```
def área_rectángulo(altura, anchura):
    return altura * anchura
```

Observa que los diferentes parámetros de una función deben separarse por comas. Al usar la función, los argumentos también deben separarse por comas:

```
rectangulo.py
def área_rectángulo(altura, anchura):
    return altura * anchura

print(área_rectángulo(3, 4))
```



▶ 279 Define una función que, dado el valor de los tres lados de un triángulo, devuelva la longitud de su perímetro.

▶ 280 Define una función que, dados dos parámetros b y x, devuelva el valor de $\log_b(x)$, es decir, el logaritmo en base b de x.

▶ 281 Diseña una función que devuelva la solución de la ecuación lineal $ax + b = 0$ dados a y b. Si la ecuación tiene infinitas soluciones o no tiene solución alguna, la función lo detectará y devolverá el valor None.

▶ 282 Diseña una función que calcule $\sum_{i=a}^{b} i$ dados a y b. Si a es mayor que b, la función devolverá el valor 0.

▶ 283 Diseña una función que calcule $\prod_{i=a}^{b} i$ dados a y b. Si a es mayor que b, la función devolverá el valor 0. Si 0 se encuentra entre a y b, la función devolverá también el valor cero, pero sin necesidad de iterar en un bucle.

▶ 284 Define una función llamada *raíz_n_ésima* que devuelva el valor de $\sqrt[n]{x}$. (Nota: recuerda que $\sqrt[n]{x}$ es $x^{1/n}$).

▶ 285 Haz una función que reciba un número de DNI y una letra. La función devolverá True si la letra corresponde a ese número de DNI, y False en caso contrario. La función debe llamarse *comprueba_letra_dni*.

Si lo deseas, puedes llamar a la función *letra_dni*, desarrollada en el ejercicio 260, desde esta nueva función.

▶ 286 Diseña una función que diga (mediante la devolución de True o False) si dos números son *amigos*. Dos números son amigos si la suma de los divisores del primero (excluído él) es igual al segundo y viceversa.

6.2.3. Definición y uso de funciones sin parámetros

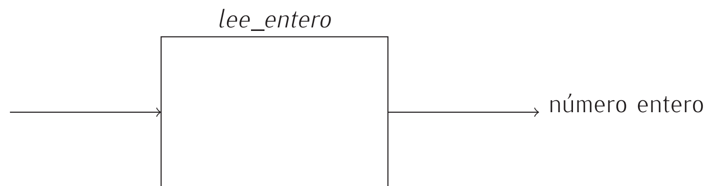
Vamos a considerar ahora cómo definir e invocar funciones sin parámetros. En realidad hay poco que decir: lo único que debes tener presente es que es obligatorio poner paréntesis a continuación del identificador, tanto al definir la función como al invocarla.

En el siguiente ejemplo se define y usa una función que lee de teclado un número entero:

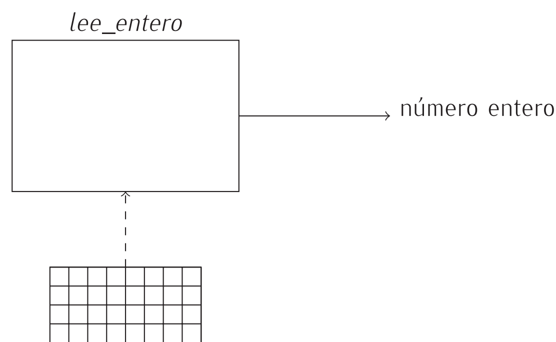
```
lee_entero.py
def lee_entero():
    return int(input())

a = lee_entero()
```

Recuerda: al llamar a una función los paréntesis *no* son opcionales. Podemos representar esta función como una caja que proporciona un dato de salida sin ningún dato de entrada:



Mmmm. Te hemos dicho que la función no recibe dato alguno y debes estar pensando que te hemos engañado, pues la función lee un dato de teclado. Quizá este diagrama represente mejor la entrada/salida de la función:



De acuerdo; pero no te equivoques: el dato leído de teclado no es un dato que el programa suministre a la función mediante un parámetro.

Parámetros o teclado

Un error frecuente al diseñar funciones consiste en tratar de obtener la información directamente de teclado. No es que esté prohibido, pero es ciertamente excepcional que una función obtenga la información de ese modo. Cuando te pidan diseñar una función que recibe uno o más datos, se sobreentiende que debes suministrarlos como argumentos en la llamada, no leerlos de teclado. Cuando queramos que la función lea algo de teclado, lo diremos *explícitamente*.

Insistimos, y esta vez ilustrando el error con un ejemplo. Imagina que te piden que diseñes una función que diga si un número es par devolviendo True si es así y False en caso contrario. Te piden una función como esta:

```
def es_par(n):
    return n % 2 == 0
```

Muchos programadores novatos escriben *erróneamente* una función como esta otra:

```
def es_par():
    n = int(input('Dame␣un␣número:␣'))
    return n % 2 == 0
```

Está mal. Escribir esa función así demuestra, cuando menos, falta de soltura en el diseño de funciones. Si hubiésemos querido una función como esa, te hubiésemos pedido una función que lea de teclado un número entero y devuelva True si es par y False en caso contrario.

Esta otra función lee un número de teclado y se asegura de que sea positivo:

```
lee_positivo.py
def lee_entero_positivo():
    número = int(input())
    while número < 0:
        número = int(input())
    return número

a = lee_entero_positivo()
```

Y esta versión muestra por pantalla un mensaje informativo cuando el usuario se equivoca:

lee_positivo.py
```
def lee_entero_positivo():
    número = int(input())
    while número < 0:
        print('Ha␣cometido␣un␣error:␣el␣número␣debe␣ser␣positivo.')
        número = int(input())
    return número

a = lee_entero_positivo()
```

Los paréntesis son necesarios

Un error típico de los aprendices es llamar a las funciones sin parámetros omitiendo los paréntesis, pues les parecen innecesarios. Veamos qué ocurre en tal caso:

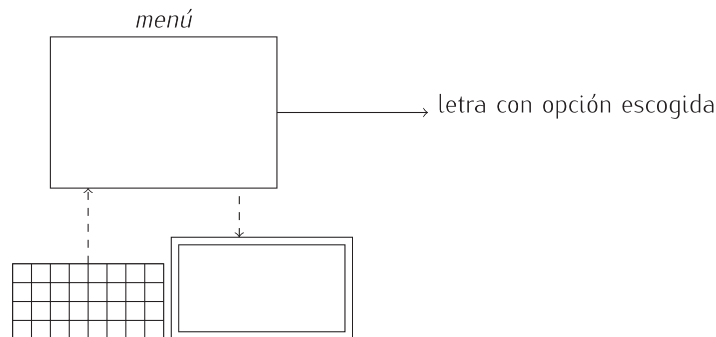
```
>>> def saluda():↵
...     print('Hola')↵
... ↵
>>> saluda()↵
Hola
>>> saluda↵
<function saluda at 0x364c7c0>
```

Como puedes ver, el último resultado no es la impresión del mensaje «Hola», sino otro encerrado entre símbolos de menor y mayor. Estamos llamando incorrectamente a la función: *saluda*, sin paréntesis, es un «objeto» Python ubicado en la dirección de memoria que se muestra en hexadecimal (número que puede ser distinto con cada ejecución).

Ciertas técnicas avanzadas de programación sacan partido del uso del identificador de la función sin paréntesis, pero aún no estás preparado para entender cómo y por qué. El cuadro «Un método de integración genérico» te proporcionará más información.

Una posible aplicación de la definición de funciones sin argumentos es la presentación de menús con selección de opción por teclado. Esta función, por ejemplo, muestra un menú con tres opciones, pide al usuario que seleccione una y se asegura de que la opción seleccionada es válida. Si el usuario se equivoca, se le informa por pantalla del error:

funcion_menu.py
```
def menú():
    opción = ''
    while not (opción >= 'a' and opción <= 'c'):
        print('Cajero␣automático.')
        print('a)␣Ingresar␣dinero.')
        print('b)␣Sacar␣dinero.')
        print('c)␣Consultar␣saldo.')
        opción = input('Escoja␣una␣opción:␣')
        if not (opción >= 'a' and opción <= 'c'):
            print('Solo␣puede␣escoger␣a,␣b␣o␣c.␣Inténtelo␣de␣nuevo.')
    return opción
```



Hemos dibujado una pantalla para dejar claro que uno de los cometidos de esta función es mostrar información por pantalla (las opciones del menú).

Si en nuestro programa principal se usa con frecuencia el menú, bastará con efectuar las correspondientes llamadas a la función *menú*() y almacenar la opción seleccionada en una variable. Así:

```
acción = menú()
```

La variable *acción* contendrá la letra seleccionada por el usuario. Gracias al control que efectúa la función, estaremos seguros de que dicha variable contiene una 'a', una 'b' o una 'c'.

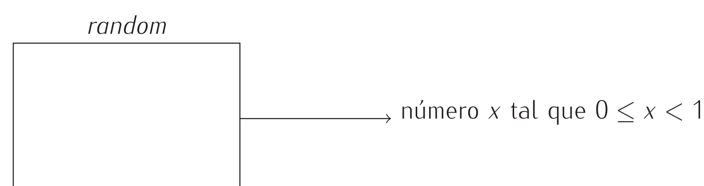
▶ 287 ¿Funciona esta otra versión de *menú*?

funcion_menu.py

```
def menú():
    opción = ''
    while len(opción) != 1 or opción not in 'abc':
        print('Cajero automático.')
        print('a) Ingresar dinero.')
        print('b) Sacar dinero.')
        print('c) Consultar saldo.')
        opción = input('Escoja una opción: ')
        if len(opción) != 1 or opción not in 'abc':
            print('Solo puede escoger a, b o c. Inténtelo de nuevo.')
    return opción
```

▶ 288 En un programa que estamos diseñando preguntamos al usuario numerosas cuestiones que requieren una respuesta afirmativa o negativa. Diseña una función llamada *sí_o_no* que reciba una cadena (la pregunta). Dicha cadena se mostrará por pantalla y se solicitará al usuario que responda. Solo aceptaremos como respuestas válidas 'sí', 's', 'Sí', 'SÍ', 'no', 'n', 'No', 'NO', las cuatro primeras para respuestas afirmativas y las cuatro últimas para respuestas negativas. Cada vez que el usuario se equivoque, en pantalla aparecerá un mensaje que le recuerde las respuestas aceptables. La función devolverá True si la respuesta es afirmativa, y False en caso contrario.

Hay funciones sin parámetros que puedes importar de módulos. Una que usaremos en varias ocasiones es *random* (en inglés «random» significa «aleatorio»). La función *random*, definida en el módulo que tiene el mismo nombre, devuelve un número al azar mayor o igual que 0.0 y menor que 1.0.



Veamos un ejemplo de uso de la función:

```
>>> from random import random↵
>>> random()↵
0.938605082516412
>>> random()↵
0.7512660740045009
>>> random()↵
0.0890845295450503
>>> random()↵
0.3934959857245368
```

¿Ves? La función se invoca sin argumentos (entre los paréntesis no hay nada) y cada vez que lo hacemos obtenemos un resultado diferente. ¿Qué interés tiene una función tan extraña? Una función capaz de generar números aleatorios encuentra muchos campos de aplicación: estadística, videojuegos, simulación, etc. Dentro de poco le sacaremos partido.

▶ 289 Diseña una función sin argumentos que devuelva un número aleatorio mayor o igual que 0.0 y menor que 10.0. Puedes llamar a la función *random* desde tu función.

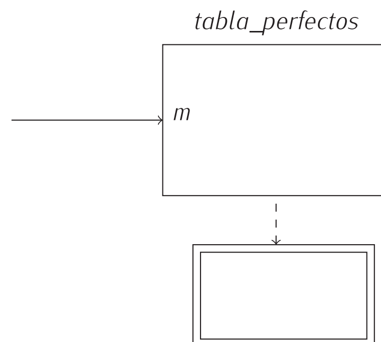
▶ 290 Diseña una función sin argumentos que devuelva un número aleatorio mayor o igual que −10.0 y menor que 10.0.

▶ 291 Para diseñar un juego de tablero nos vendrá bien disponer de un «dado electrónico». Escribe una función Python sin argumentos llamada *dado* que devuelva un número *entero* aleatorio entre 1 y 6.

6.2.4. Procedimientos: funciones sin devolución de valor

No todas las funciones devuelven un valor. Una función que no devuelve un valor se denomina *procedimiento*. ¿Y para qué sirve una función que no devuelve nada? Bueno, puede, por ejemplo, mostrar mensajes o resultados por pantalla. No te equivoques: *mostrar* algo por pantalla no es *devolver* nada. Mostrar un mensaje por pantalla es un *efecto secundario*.

Veámoslo con un ejemplo. Vamos a implementar ahora un programa que solicita al usuario un número y muestra por pantalla todos los números perfectos entre 1 y dicho número.



Reutilizaremos la función *es_perfecto* que definimos antes en este mismo capítulo. Como la solución no es muy complicada, te la ofrecemos completamente desarrollada:

tabla_perfectos.py
```
def es_perfecto(n): # Averigua si el número n es o no es perfecto.
    sumatorio = 0
    for i in range(1, n):
        if n % i == 0:
            sumatorio += i
    return sumatorio == n

def tabla_perfectos(m): # Muestra todos los números perfectos entre 1 y m.
```

```
9      for i in range(1, m+1):
10         if es_perfecto(i):
11             print(i, 'es␣un␣número␣perfecto')
12
13 número = int(input('Dame␣un␣número:␣'))
14 tabla_perfectos(número)
```

Fíjate en que la función *tabla_perfectos* no devuelve nada (no hay sentencia **return**): es un procedimiento. También resulta interesante la línea 10: como *es_perfecto* devuelve True o False, podemos utilizarla directamente como condición del **if**.

Condicionales que trabajan directamente con valores lógicos

Ciertas funciones devuelven directamente un valor lógico. Considera, por ejemplo, esta función, que nos dice si un número es o no es par:

```
def es_par(n):
    return n % 2 == 0
```

Si una sentencia condicional toma una decisión en función de si un número es par o no, puedes codificar así la condición:

```
if es_par(n):
    ...
```

Observa que no hemos usado comparador alguno en la condición del **if**. ¿Por qué? Porque la función *es_par*(*n*) devuelve True o False directamente. Los programadores primerizos tienen tendencia a codificar la misma condición así:

```
if es_par(n) == True:
    ...
```

Es decir, comparan el valor devuelto por *es_par* con el valor True, pues les da la sensación de que un **if** sin comparación no está completo. No pasa nada si usas la comparación, pero es innecesaria. Es más, si no usas la comparación, el programa es más legible: la sentencia condicional se lee directamente como «si *n* es par» en lugar de «si *n* es par es cierto», que es un extraño circunloquio.

Si deseas comprobar que el número es impar, puedes hacerlo así:

```
if not es_par(n):
    ...
```

Es muy legible: «si no es par *n*». Los programadores que están empezando escriben:

```
if es_par(n) == False:
    ...
```

que se lee como «si *n* es par es falso». Peor, ¿no?

Acostúmbrate a usar la versión que no usa operador de comparación. Es más legible.

▶ 292 Diseña un programa que, dado un número *n*, muestre por pantalla todas las parejas de números amigos menores que *n*. La impresión de los resultados debe hacerse desde un procedimiento.

Dos números amigos solo deberán aparecer una vez por pantalla. Por ejemplo, 220 y 284 son amigos: si aparece el mensaje «`220 y 284 son amigos`», no podrá aparecer el mensaje «`284 y 220 son amigos`», pues es redundante.

Debes diseñar una función que diga si dos números son amigos y un procedimiento que muestre la tabla.

▶ 293 Implementa un procedimiento Python tal que, dado un número entero, muestre por

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

pantalla sus cifras en orden inverso. Por ejemplo, si el procedimiento recibe el número 324, mostrará por pantalla el 4, el 2 y el 3 (en líneas diferentes).

▶ 294 Diseña una función *es_primo* que determine si un número es primo (devolviendo True) o no (devolviendo False). Diseña a continuación un procedimiento *muestra_primos* que reciba un número y muestre por pantalla todos los números primos entre 1 y dicho número.

¿Y qué ocurre si utilizamos un procedimiento como si fuera una función con devolución de valor? Podemos hacer la prueba. Asignemos a una variable el resultado de llamar a *tabla_perfectos* y mostremos por pantalla el valor de la variable:

tabla_perfectos.py
```
def es_perfecto(n): # Averigua si el número n es o no es perfecto.
    sumatorio = 0
    for i in range(1, n):
        if n % i == 0:
            sumatorio += i
    return sumatorio == n

def tabla_perfectos(m): # Muestra todos los números perfectos entre 1 y m.
    for i in range(1, m+1):
        if es_perfecto(i):
            print(i, 'es␣un␣número␣perfecto')

resultado = tabla_perfectos(100)
print(resultado)
```

Por pantalla aparece lo siguiente:

```
6 es un número perfecto
28 es un número perfecto
None
```

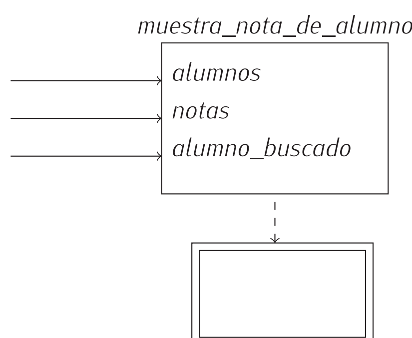
Mira la última línea, que muestra el contenido de resultado. Recuerda que Python usa None para indicar un valor nulo o la ausencia de valor, y una función que «no devuelve nada» devuelve la «ausencia de valor», ¿no?

Cambiamos de tercio. Supón que mantenemos dos listas con igual número de elementos. Una de ellas, llamada *alumnos*, contiene una serie de nombres y la otra, llamada *notas*, una serie de números flotantes entre 0.0 y 10.0. En *notas* guardamos la calificación obtenida por los alumnos cuyos nombres están en *alumnos*: la nota *notas*[*i*] corresponde al estudiante *alumnos*[*i*]. Una posible configuración de las listas sería esta:

```
alumnos = ['Ana␣Pi', 'Pau␣López', 'Luis␣Sol', 'Mar␣Vega', 'Paz␣Mir']
notas   = [10, 5.5, 2.0, 8.5, 7.0]
```

De acuerdo con ella, el alumno Pau López, por ejemplo, fue calificado con un 5.5.

Nos piden diseñar un procedimiento que recibe como datos las dos listas y una cadena con el nombre de un estudiante. Si el estudiante pertenece a la clase, el procedimiento imprimirá su nombre y nota en pantalla. Si no es un alumno incluido en la lista, se imprimirá un mensaje que lo advierta.



Índice

Valor de retorno o pantalla

Te hemos mostrado de momento que es posible imprimir información directamente por pantalla desde una función (o procedimiento). Ojo: solo lo hacemos cuando el propósito de la función es mostrar esa información. Muchos aprendices que no han comprendido bien el significado de la sentencia **return**, la sustituyen por una sentencia *print*. Mal. Cuando te piden que diseñes una función que *devuelva* un valor, te piden que lo haga con la sentencia **return**, que es la única forma válida (que conoces) de devolver un valor. Mostrar algo por pantalla no es devolver ese algo. Cuando quieran que muestres algo por pantalla, te lo dirán explícitamente.

Supón que te piden que diseñes una función que reciba un entero y devuelva su última cifra. Te piden esto:

```
def última_cifra(número):
    return número % 10
```

No te piden esto otro:

```
def última_cifra(número):
    print(número % 10)
```

Fíjate en que la segunda definición hace que la función no pueda usarse en expresiones como esta:

```
a = última_cifra(10293) + 1
```

Como *última_cifra* no *devuelve* nada, ¿qué valor se está sumando a 1 y guardando en *a*?

¡Ah! Aún se puede hace peor. Hay quien define la función así:

```
def última_cifra():
    número = int(input('Dame un número: '))
    print(número % 10)
```

No solo demuestra no entender qué es el valor de retorno; además, demuestra que no tiene ni idea de lo que es el paso de parámetros. Evita dar esa impresión: lee bien lo que se pide y usa parámetros y valor de retorno a menos que se te diga explícitamente lo contrario. Lo normal es que la mayor parte de las funciones produzcan datos (devueltos con **return**) a partir de otros datos (obtenidos con parámetros) y que el programa principal o funciones muy específicas lean de teclado y muestren por pantalla.

Aquí tienes una primera versión:

clase.py
```
def muestra_nota_de_alumno(alumnos, notas, alumno_buscado):
    encontrado = False
    for i in range(len(alumnos)):
        if alumnos[i] == alumno_buscado:
            print(alumno_buscado, notas[i])
            encontrado = True
    if not encontrado:
        print('El alumno {0} no pertenece al grupo'.format(alumno_buscado))
```

Lo podemos hacer más eficientemente: cuando hemos encontrado al alumno e impreso el correspondiente mensaje, no tiene sentido seguir iterando:

clase.py
```
def muestra_nota_de_alumno(alumnos, notas, alumno_buscado):
    encontrado = False
    for i in range(len(alumnos)):
        if alumnos[i] == alumno_buscado:
            print(alumno_buscado, notas[i])
            encontrado = True
```

```
            break
    if not encontrado:
        print('El alumno {0} no pertenece al grupo'.format(alumno_buscado))
```

Esta otra versión es aún más breve[3]:

clase.py

```
def muestra_nota_de_alumno(alumnos, notas, alumno_buscado):
    for i in range(len(alumnos)):
        if alumnos[i] == alumno_buscado:
            print(alumno_buscado, notas[i])
            return
    print('El alumno {0} no pertenece al grupo'.format(alumno_buscado))
```

Los procedimientos aceptan el uso de la sentencia **return** aunque, eso sí, sin expresión alguna a continuación (recuerda que los procedimientos no devuelven valor alguno). ¿Qué hace esa sentencia? Aborta inmediatamente la ejecución de la llamada a la función. Es, en cierto modo, similar a una sentencia **break** en un bucle, pero asociada a la ejecución de una función.

▶ 295 En el problema de los alumnos y las notas, se pide:

1) Diseñar un *procedimiento* que reciba las dos listas y muestre por pantalla el nombre de todos los estudiantes que aprobaron el examen.

2) Diseñar una *función* que reciba la lista de notas y devuelva el número de aprobados.

3) Diseñar un *procedimiento* que reciba las dos listas y muestre por pantalla el nombre de todos los estudiantes que obtuvieron la máxima nota.

4) Diseñar un *procedimiento* que reciba las dos listas y muestre por pantalla el nombre de todos los estudiantes cuya calificación es igual o superior a la calificación media.

5) Diseñar una *función* que reciba las dos listas y un nombre (una cadena); si el nombre está en la lista de estudiantes, devolverá su nota, si no, devolverá None.

▶ 296 Tenemos los tiempos de cada ciclista y etapa para los participantes en la última vuelta ciclista local. La lista *ciclistas* contiene una serie de nombres. La matriz *tiempos* tiene una fila por cada ciclista, en el mismo orden con que aparecen en *ciclistas*. Cada fila tiene el tiempo en segundos (un valor flotante) invertido en cada una de las 5 etapas de la carrera. ¿Complicado? Quizás te ayude este ejemplo de lista *ciclistas* y de matriz *tiempos* para 3 corredores.

```
ciclistas = ['Pere Porcar', 'Joan Beltran', 'Lledó Fabra']
tiempo = [[10092.0, 12473.1, 13732.3, 10232.1, 10332.3],
          [11726.2, 11161.2, 12272.1, 11292.0, 12534.0],
          [10193.4, 10292.1, 11712.9, 10133.4, 11632.0]]
```

En el ejemplo, el ciclista Joan Beltran invirtió 11161.2 segundos en la segunda etapa.
Se pide:

- Una función que reciba la lista y la matriz y devuelva el ganador de la vuelta (aquel cuya suma de tiempos en las 5 etapas es mínima).
- Una función que reciba la lista, la matriz y un número de etapa y devuelva el nombre del ganador de la etapa.
- Un procedimiento que reciba la lista, la matriz y muestre por pantalla el ganador de cada una de las etapas.

---

[3]… aunque puede disgustar a los puristas de la programación estructurada. Según estos, solo debe haber un punto de salida de la función: el final de su cuerpo. Salir directamente desde un bucle les parece que dificulta la comprensión del programa.

6.2.5. Funciones que devuelven varios valores mediante una lista

En principio una función puede devolver un solo valor con la sentencia **return**. Pero sabemos que una lista es un objeto que contiene una secuencia de valores. Si devolvemos una lista podemos, pues, devolver varios valores.

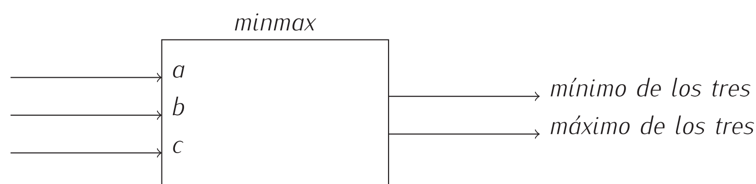
Por ejemplo, una función puede devolver al mismo tiempo el mínimo y el máximo de 3 números:

minmax.py
```python
def minmax(a, b, c):
    # Calcular el mínimo
    if a < b:
        if a < c:
            min = a
        else:
            min = c
    else:
        if b < c:
            min = b
        else:
            min = c

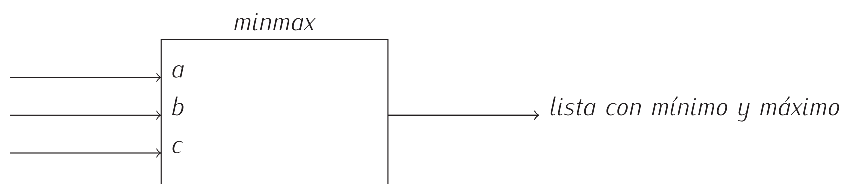
    # Calcular el máximo
    if a > b:
        if a > c:
            max = a
        else:
            max = c
    else:
        if b > c:
            max = b
        else:
            max = c

    return [min, max]
```

Podemos representar a la función con este diagrama:



aunque quizá sea más apropiado este otro:



¿Cómo podríamos llamar a esa función? Una posibilidad es esta:

minmax.py
```python
def minmax(a, b, c):
    # Calcular el mínimo
    if a < b:
        if a < c:
            min = a
```

```
        else:
            min = c
    else:
        if b < c:
            min = b
        else:
            min = c

    # Calcular el máximo
    if a > b:
        if a > c:
            max = a
        else:
            max = c
    else:
        if b > c:
            max = b
        else:
            max = c

    return [min, max]

a = minmax(10, 2, 5)
print('El␣mínimo␣es', a[0])
print('El␣máximo␣es', a[1])
```

Y esta es otra:

minmax.py
```
def minmax(a, b, c):
    # Calcular el mínimo
    if a < b:
        if a < c:
            min = a
        else:
            min = c
    else:
        if b < c:
            min = b
        else:
            min = c

    # Calcular el máximo
    if a > b:
        if a > c:
            max = a
        else:
            max = c
    else:
        if b > c:
            max = b
        else:
            max = c

    return [min, max]

[mínimo, máximo] = minmax(10, 2, 5)
print('El␣mínimo␣es', mínimo)
print('El␣máximo␣es', máximo)
```

En este segundo caso hemos asignado una lista a otra. ¿Qué significa eso para Python?

Pues que cada elemento de la lista a la derecha del igual debe asignarse a cada variable de la lista a la izquierda del igual.

▶ 297 ¿Qué aparecerá por pantalla al ejecutar este programa?

```
a = 1
b = 2
[a, b] = [b, a]
print(a, b)
```

▶ 298 Diseña una función que reciba una lista de enteros y devuelva los números mínimo y máximo de la lista simultáneamente.

▶ 299 Diseña una función que reciba los tres coeficientes de una ecuación de segundo grado de la forma $ax^2 + bx + c = 0$ y devuelva una lista con sus soluciones reales. Si la ecuación solo tiene una solución real, devuelve una lista con dos copias de la misma. Si no tiene solución real alguna o si tiene infinitas soluciones, devuelve una lista con dos copias del valor None.

▶ 300 Diseña una función que reciba una lista de palabras (cadenas) y devuelva, simultáneamente, la primera y la última palabras según el orden alfabético.

Inicialización múltiple e intercambio

Ahora que sabes que es posible asignar valores a varias variables simultáneamente, puedes simplificar algunos programas que empiezan con la inicialización de varias variables. Por ejemplo, esta serie de asignaciones:

```
a = 1
b = 2
c = 3
```

puede reescribirse así:

```
[a, b, c] = [1, 2, 3]
```

Mmmm. Aún podemos escribirlo más brevemente:

```
a, b, c = 1, 2, 3
```

¿Por qué no hacen falta los corchetes? Porque en este caso estamos usando una estructura ligeramente diferente: una *tupla*. Una tupla es una lista inmutable y no necesita ir encerrada entre corchetes.

Así pues, el intercambio del valor de dos variables puede escribirse así:

```
a, b = b, a
```

Cómodo, ¿no crees?

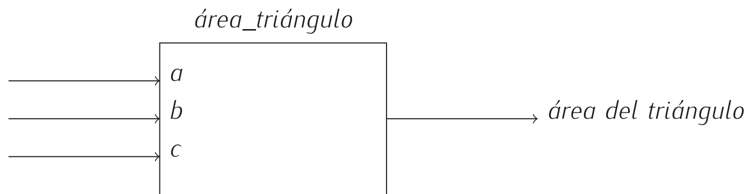
6.3. Variables locales y variables globales

Observa que en el cuerpo de las funciones es posible definir y usar variables. Vamos a estudiar con detenimiento algunas propiedades de las variables definidas en el cuerpo de una función y en qué se diferencian de las variables que definimos fuera de cualquier función, es decir, en el denominado programa principal.

Empecemos con un ejemplo. Definamos una función que, dados los tres lados de un triángulo, devuelva el valor de su área. Recuerda que si a, b y c son dichos lados, el área del triángulo es

$$\sqrt{s(s-a)(s-b)(s-c)},$$

donde $s = (a + b + c)/2$.



La función se define así:

```
triangulo.py
1 from math import sqrt
2
3 def área_triángulo(a, b, c):
4     s = (a + b + c) / 2
5     return sqrt(s * (s-a) * (s-b) * (s-c))
```

La línea 4, en el cuerpo de la función, define la variable s asignándole un valor que es instrumental para el cálculo del área del triángulo, es decir, que no nos interesa por sí mismo, sino por ser de ayuda para obtener el valor que realmente deseamos calcular: el que resulta de evaluar la expresión de la línea 5.

La función *área_triángulo* se usa como cabe esperar:

```
triangulo.py
1 from math import sqrt
2
3 def área_triángulo(a, b, c):
4     s = (a + b + c) / 2
5     return sqrt(s * (s-a) * (s-b) * (s-c))
6
7 print(área_triángulo(1, 3, 2.5))
```

Ahora viene lo importante: la variable *s solo existe en el cuerpo de la función*. Fuera de dicho cuerpo, s no está definida. El siguiente programa provoca un error al ejecutarse porque intenta acceder a s desde el programa principal:

```
triangulo.py
1 from math import sqrt
2
3 def área_triángulo(a, b, c):
4     s = (a + b + c) / 2
5     return sqrt(s * (s-a) * (s-b) * (s-c))
6
7 print(área_triángulo(1, 3, 2.5))
8 print(s)
```

Cuando se ejecuta ocurre esto:

```
1.1709371246996996
Traceback (most recent call last):
  File "triangulo.py", line 8, in <module>
    print(s)
NameError: name 's' is not defined
```

La primera línea mostrada en pantalla es el resultado de ejecutar la línea 7 del programa. La línea 7 incluye una llamada a *área_triángulo*, así que el flujo de ejecución ha pasado por la

línea 4 y *s* se ha creado correctamente. De hecho, se ha accedido a su valor en la línea 5 y no se ha producido error alguno. Sin embargo, al ejecutar la línea 8 se ha producido un error por intentar mostrar el valor de una variable inexistente: *s*. La razón es que *s* se ha creado en la línea 4 y se ha destruido tan pronto ha finalizado la ejecución de *área_triángulo*.

Las variables que solo existen en el cuerpo de una función se denominan *variables locales*. En contraposición, el resto de variables se llaman *variables globales*.

También los parámetros formales de una función se consideran variables locales, así que no puedes acceder a su valor fuera del cuerpo de la función.

Fíjate en este otro ejemplo:

```
triangulo.py
from math import sqrt

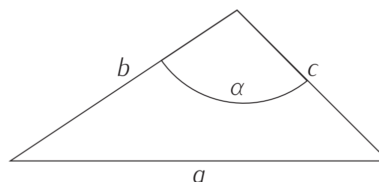
def área_triangulo(a, b, c):
    s = (a + b + c) / 2
    return sqrt(s * (s-a) * (s-b) * (s-c))

print(área_triángulo(1, 3, 2.5))
print(a)
```

¿Y cuándo se crean *a*, *b* y *c*? ¿Con qué valores? Cuando llamamos a la función con, por ejemplo, *área_triángulo*(1, 3, 2.5), ocurre lo siguiente: los parámetros *a*, *b* y *c* se crean como variables locales en la función y apuntan a los valores 1, 3 y 2.5, respectivamente. Se inicia entonces la ejecución del cuerpo de *área_triángulo* hasta llegar a la línea que contiene el **return**. El valor que resulta de evaluar la expresión que sigue al **return** se devuelve como resultado de la llamada a la función. Al acabar la ejecución de la función, las variables locales *a*, *b* y *c dejan de existir* (del mismo modo que deja de existir la variable local *s*).

Para ilustrar los conceptos de variables locales y globales con mayor detalle vamos a utilizar la función *área_triángulo* en un programa un poco más complejo.

Imagina que queremos ayudarnos con un programa en el cálculo del área de un triángulo de lados *a*, *b* y *c* y en el cálculo del ángulo α (en grados) opuesto al lado *a*.



El ángulo α se calcula con la fórmula

$$\alpha = \frac{180}{\pi} \cdot \arcsin\left(\frac{2s}{bc}\right),$$

donde *s* es el área del triángulo y arcsin es la función arco-seno. (La función matemática «arcsin» está definida en el módulo *math* con el identificador *asin*).

Analiza este programa en el que hemos destacado las diferentes apariciones del identificador *s*:

```
area_y_angulo.py
from math import sqrt, asin, pi

def área_triángulo(a, b, c):
    s = (a + b + c) / 2
    return sqrt(s * (s-a) * (s-b) * (s-c))

def ángulo_alfa(a, b, c):
    s = área_triángulo(a, b, c)
    return 180 / pi * asin(2 * s / (b*c))

```

```
11 def menú():
12     opción = 0
13     while opción != 1 and opción != 2:
14         print('1) Calcular área del triángulo')
15         print('2) Calcular ángulo opuesto al primer lado')
16         opción = int(input('Escoge opción: '))
17     return opción
18
19 lado1 = float(input('Dame lado a: '))
20 lado2 = float(input('Dame lado b: '))
21 lado3 = float(input('Dame lado c: '))
22
23 s = menú()
24
25 if s == 1:
26     resultado = área_triángulo(lado1, lado2, lado3)
27 else:
28     resultado = ángulo_alfa(lado1, lado2, lado3)
29
30 print('Escogiste la opción', s)
31 print('El resultado es:', resultado)
```

Hagamos una traza del programa para esta ejecución:

- La línea 1 importa las funciones *sqrt* (raíz cuadrada) y *asin* (arcoseno) y la variable *pi* (aproximación de π).
- Las líneas 3–5 «enseñan» a Python cómo se realiza un cálculo determinado al que denominamos *área_triángulo* y que necesita tres datos de entrada.
- Las líneas 7–9 «enseñan» a Python cómo se realiza un cálculo determinado al que denominamos *ángulo_alfa* y que también necesita tres datos de entrada.
- Las líneas 11–17 definen la función *menú*. Es una función sin parámetros cuyo cometido es mostrar un menú con dos opciones, esperar a que el usuario escoja una y devolver la opción seleccionada.
- Las líneas 19–21 leen de teclado el valor (flotante) de tres variables: *lado1*, *lado2* y *lado3*. En nuestra ejecución, las variables valdrán 5.0, 4.0 y 3.0, respectivamente.
- La línea 23 contiene una llamada a la función *menú*. En este punto, Python memoriza que se encontraba ejecutando la línea 23 cuando se produjo una llamada a función y deja su ejecución *en suspenso*. Salta entonces a la línea 12, es decir, al cuerpo de la función *menú*. Sigamos el flujo de ejecución en dicho cuerpo:
  - Se ejecuta la línea 12. La variable local *opción* almacena el valor 0.
  - En la línea 13 hay un bucle **while**. ¿Es *opción* distinto de 1 y de 2? Sí. Entramos, pues, en el bloque del bucle: la siguiente línea a ejecutar es la 14.
  - En la línea 14 se imprime un texto en pantalla (el de la primera opción).
  - En la línea 15 se imprime otro texto en pantalla (el de la segunda opción).
  - En la línea 16 se lee el valor de *opción* de teclado, que en esta ejecución es 1.
  - Como el bloque del bucle no tiene más líneas, volvemos a la línea 13. Nos volvemos a preguntar ¿es *opción* distinto de 1 y a la vez distinto de 2? No: *opción* vale 1. El bucle finaliza y saltamos a la línea 17.
  - En la línea 17 se devuelve el valor 1, que es el valor de *opción*, y la variable local *opción* se destruye.

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

- ¿Qué línea se ejecuta ahora? La ejecución de la llamada a la función ha finalizado, así que Python regresa a la línea desde la que se produjo la llamada (la línea 23), cuya ejecución había quedado en suspenso. El valor devuelto por la función (el valor 1) se almacena ahora en una variable llamada *s*.

- La línea 25 compara el valor de *s* con el valor 1 y, como son iguales, la siguiente línea a ejecutar es la 26 (las líneas 27 y 28 no se ejecutarán).

- La línea 26 asigna a *resultado* el resultado de invocar a *área_triángulo* con los valores 5.0, 4.0 y 3.0. Al invocar la función, el flujo de ejecución del programa «salta» a su cuerpo y la ejecución de la línea 26 queda *en suspenso*.

    - Saltamos, pues, a la línea 4, con la que empieza el cuerpo de la función *área_triángulo*. ¡Ojo!, los parámetros *a*, *b* y *c* se crean como variables locales y toman los valores 5.0, 4.0 y 3.0, respectivamente (son los valores de *lado1*, *lado2* y *lado3*). En la línea 4 se asigna a *s*, una nueva variable local, el valor que resulte de evaluar `(`*a* `+` *b* `+` *c*`)/2`, es decir, 6.0.
    - En la línea 5 se devuelve el resultado de evaluar *sqrt*`(`*s* `* (`*s*`-`*a*`) * (`*s*`-`*b*`) * (`*s*`-`*c*`))`, que también es, casualmente, 6.0. Tanto *s* como los tres parámetros dejan de existir.

- Volvemos a la línea 26, cuya ejecución estaba suspendida a la espera de conocer el valor de la llamada a *área_triángulo*. El valor devuelto, 6.0, se asigna a *resultado*.

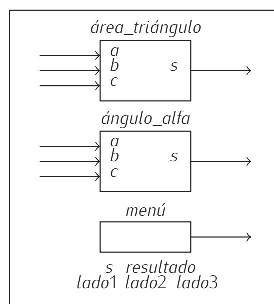
- La línea 30 muestra por pantalla el valor actual de *s*… ¿y qué valor es ese? ¡Al ejecutar la línea 23 le asignamos a *s* el valor 1, pero al ejecutar la línea 4 le asignamos el valor 6.0! ¿Debe salir por pantalla, pues, un 6.0? No: la línea 23 asignó el valor 1 a la *variable global s*. El 6.0 de la línea 4 se asignó a la *variable s local a la función área_triángulo*, que ya no existe.

- Finalmente, el valor de *resultado* se muestra por pantalla en la línea 31.

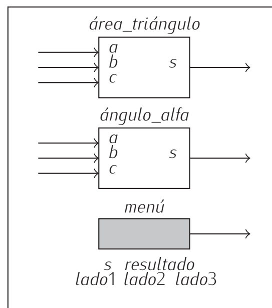
Observa que llamamos *s* a dos variables diferentes y que cada una de ellas «recuerda» su valor sin interferir con el valor de la otra. Si accedemos a *s* desde *área_triángulo*, accedemos a la *s* local a *área_triángulo*. Si accedemos a *s* desde fuera de cualquier función, accedemos a la *s* global.

Puede que te parezca absurdo que Python distinga entre variables locales y variables globales, pero lo cierto es que disponer de estos dos tipos de variable es de gran ayuda. Piensa en qué ocurriría si la variable *s* de la línea 4 fuese global: al acabar la ejecución de *área_triángulo*, *s* recordaría el valor 6.0 y habría olvidado el valor 1. El texto impreso en la línea 30 sería erróneo, pues se leería así: «`Escogiste la opción 6.0000`». Disponer de variables locales permite asegurarse de que las llamadas a función no modificarán accidentalmente nuestras variables globales, aunque se llamen igual.

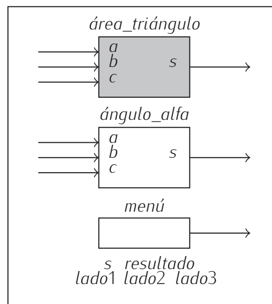
La siguiente figura ilustra la idea de que cada elemento del programa tiene un identificador que lo hace accesible o visible desde un *entorno* o *ámbito* diferente.



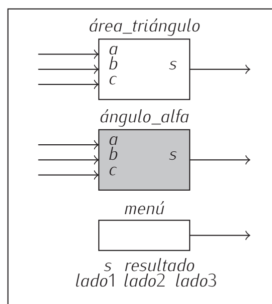
Cada función define un *ámbito local* propio: su cuerpo. Los identificadores de las variables locales solo son visibles en su ámbito local. Por ejemplo, la variable *opción* definida en la función *menú* solo es visible en el cuerpo de *menú*. En este diagrama marcamos en tono gris la región en la que es visible esa variable:



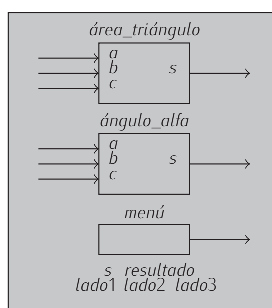
Fuera de la zona gris, tratar de acceder al valor de *opción* se considera un error. ¿Qué pasa con las variables o parámetros de nombre idéntico definidas en *área_triángulo* y *ángulo_alfa*? Considera, por ejemplo, el parámetro *a* o la variable *s* definida en *área_triángulo*: solo es accesible desde el cuerpo de *área_triángulo*.



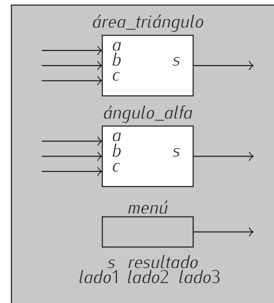
No hay confusión posible: cuando accedes al valor de *a* en el cuerpo de *área_triángulo*, accedes a su parámetro *a*. Lo mismo ocurre con la variable *s* o el parámetro *a* de *ángulo_alfa*: si se usan en el cuerpo de la función, Python sabe que nos referimos a esas variables locales:



Hay un *ámbito global* que incluye a aquellas líneas del programa que no forman parte del cuerpo de una función. Los identificadores de las variables globales son visibles en el ámbito global *y desde cualquier ámbito local*. Las variables *resultado* o *lado1*, por ejemplo, son accesibles desde cualquier punto del programa (esté dentro o fuera del cuerpo de una función). Podemos representar así su «zona de visibilidad», es decir, su ámbito:



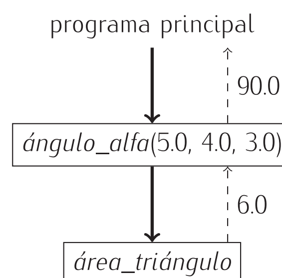
Hay una excepción a la regla de que las variables del ámbito global sean accesibles desde cualquier punto del programa: si el identificador de una variable (o función) definida en el ámbito global se usa para nombrar una variable local en una función, la variable (o función) global queda «oculta» y no es accesible desde el cuerpo de la función. Por ejemplo, la variable local *s* definida en la línea 4 hace que la variable global *s* definida en la línea 23 no sea visible en el cuerpo de la función *área_triángulo*. Su ámbito se reduce a esta región sombreada:



En el programa, la función *ángulo_alfa* presenta otro aspecto de interés: desde ella se llama a la función *área_triángulo*. El cuerpo de una función puede incluir llamadas a otras funciones. ¿Qué ocurre cuando efectuamos una llamada a *ángulo_alfa*? Supongamos que al ejecutar el programa introducimos los valores 5, 4 y 3 para *lado1*, *lado2* y *lado3* y que escogemos la opción 2 del menú. Al ejecutarse la línea 28 ocurre lo siguiente:

- Al evaluar la parte derecha de la asignación de la línea 28 se invoca la función *ángulo_alfa* con los argumentos 5, 4 y 3, con lo que la ejecución salta a la línea 8 y *a*, *b* y *c* toman los valores 5, 4 y 3, respectivamente. Python recuerda que al acabar de ejecutar la llamada, debe seguir con la ejecución de la línea 28.
    - Se ejecuta la línea 8 y, al evaluar la parte derecha de su asignación, se invoca la función *área_triángulo* con los argumentos 5, 4 y 3 (que son los valores de *a*, *b* y *c*). La ejecución salta, pues, a la línea 4 y Python recuerda que, cuando acabe de ejecutar esta nueva llamada, regresará a la línea 8.
        - En la línea 4 la variable *s* local a *área_triángulo* vale 6.0. Los parámetros *a*, *b* y *c* son *nuevas* variables locales con valores 5, 4, y 3, respectivamente.
        - Se ejecuta la línea 5 y se devuelve el resultado, que es 6.0.
    - Regresamos a la línea 8, cuya ejecución había quedado suspendida a la espera de conocer el resultado de la llamada a *área_triángulo*. Como el resultado es 6.0, se asigna dicho valor a la variable *s* local a *ángulo_alfa*. Se ejecuta la línea 9 y se devuelve el resultado de evaluar la expresión, que es 90.0.
- Sigue la ejecución en la línea 28, que había quedado en suspenso a la espera de conocer el valor de la llamada a *ángulo_alfa*. Dicho valor se asigna a *resultado*.
- Se ejecutan las líneas 30 y 31.

Podemos representar gráficamente las distintas activaciones de función mediante el denominado *árbol de llamadas*. He aquí el árbol correspondiente al último ejemplo:



Las llamadas se producen de arriba a abajo y siempre desde la función de la que parte la flecha con trazo sólido. La primera flecha parte del «programa principal» (fuera de cualquier función). El valor devuelto por cada función aparece al lado de la correspondiente flecha de trazo discontinuo.

▶ 301 Haz una traza de *area_y_angulo.py* al solicitar el valor del ángulo opuesto al lado de longitud 5 en un triángulo de lados con longitudes 5, 4 y 3.

▶ 302 ¿Qué aparecerá por pantalla al ejecutar el siguiente programa?

triangulo.py
```
from math import sqrt

def área_triángulo(a, b, c):
    s = (a + b + c) / 2
    return sqrt(s * (s-a) * (s-b) * (s-c))

s = 4
print(área_triángulo(s-1, s, s+1))
print(s)
print(a)
```

▶ 303 La función *área_triángulo* que hemos definido puede provocar un error en tiempo de ejecución: si el argumento de la raíz cuadrada calculada en su última línea es un número negativo, se producirá un error de dominio. Haz que la función solo llame a *sqrt* si su argumento es mayor o igual que cero. Si el argumento es un número negativo, la función debe devolver el valor cero. Detecta también posibles problemas en *ángulo_alfa* y modifica la función para evitar posibles errores al ejecutar el programa.

▶ 304 Vamos a adquirir una vivienda y para eso necesitaremos una hipoteca. La cuota mensual m que hemos de pagar para amortizar una hipoteca de h euros a lo largo de n años a un interés compuesto del i por cien anual se calcula con la fórmula:

$$m = \frac{hr}{1 - (1 + r)^{-12n}},$$

donde $r = i/(100 \cdot 12)$. Define una función que calcule la cuota (redondeada a dos decimales) dados h, n e i. Utiliza cuantas variables locales consideres oportuno, pero al menos r debe aparecer en la expresión cuyo valor se devuelve y antes debe calcularse y almacenarse en una variable local.

Nota: puedes comprobar la validez de tu función sabiendo que hay que pagar la cantidad de 1.166,75 € al mes para amortizar una hipoteca de 150.000 € en 15 años a un interés del 4,75 % anual.

▶ 305 Diseña una función que nos devuelva la cantidad de euros que habremos pagado finalmente al banco si abrimos una hipoteca de h euros a un interés del i por cien en n años. Si te conviene, puedes utilizar la función que definiste en el ejercicio anterior.

Nota: con los datos del ejemplo anterior, habremos pagado un total de 210.015 €.

▶ 306 Diseña una función que nos diga qué cantidad *de intereses* (en euros) habremos pagado finalmente al banco si abrimos una hipoteca de h euros a un interés del i por cien en n años. Si te conviene, puedes utilizar las funciones que definiste en los ejercicios anteriores.

Nota: con los datos del ejemplo anterior, habremos pagado un total de $210.015 - 150.000 = 60.015$ € en intereses.

▶ 307 Diseña una función que nos diga qué tanto por cien del capital inicial deberemos pagar en intereses al amortizar completamente la hipoteca. Si te conviene, puedes utilizar las funciones que definiste en los ejercicios anteriores.

Nota: con los datos del ejemplo anterior, habremos pagado un interés total del 40,01 % (60.015 € es el 40,01 % de 150.000 €).

▶ 308 Diseña un *procedimiento* que muestre por pantalla la cuota mensual que corresponde pagar por una hipoteca para un capital de h euros al i % de interés anual durante 10, 15, 20 y 25 años. (Si te conviene, rescata ahora las funciones que diseñaste como solución de los ejercicios anteriores).

▶ 309 Diseña un *procedimiento* que muestre por pantalla el capital total pagado al banco por una hipoteca de h euros al i % de interés anual durante 10, 15, 20 y 25 años. (Si te conviene, rescata ahora las funciones que diseñaste como solución de los ejercicios anteriores).

Las variables locales también pueden contener valores secuenciales. Estudiemos un ejemplo de función con una variable local de tipo secuencial: una función que recibe una lista y devuelve otra cuyos elementos son los de la primera, pero sin repetir ninguno; es decir, si la función recibe la lista `[1, 2, 1, 3, 2]`, devolverá la lista `[1, 2, 3]`.

Empecemos por definir el cuerpo de la función:

sin_repetidos.py
```
def sin_repetidos(lista):
    ...
```

¿Cómo procederemos? Una buena idea consiste en disponer de una nueva lista auxiliar (una variable local) inicialmente vacía en la que iremos insertando los elementos de la lista resultante. Podemos recorrer la lista original elemento a elemento y preguntar a cada uno de ellos si ya se encuentra en la lista auxiliar. Si la respuesta es negativa, lo añadiremos a la lista:

sin_repetidos.py
```
def sin_repetidos(lista):
    resultado = []
    for elemento in lista:
        if elemento not in resultado:
            resultado.append(elemento)
    return resultado
```

Fácil, ¿no? La variable *resultado* es local, así que su tiempo de vida se limita al de la ejecución del cuerpo de la función cuando esta sea invocada. El contenido de *resultado* se devuelve con la sentencia **return**, así que sí será accesible desde fuera. Aquí tienes un ejemplo de uso:

sin_repetidos.py
```
def sin_repetidos(lista):
    resultado = []
    for elemento in lista:
        if elemento not in resultado:
            resultado.append(elemento)
    return resultado

una_lista = sin_repetidos([1, 2, 1, 3, 2])
print(una_lista)
```

▶ 310 Diseña una función que reciba dos listas y devuelva los elementos comunes a ambas, sin repetir ninguno (intersección de conjuntos).

Ejemplo: si recibe las listas `[1, 2, 1]` y `[2, 3, 2, 4]`, devolverá la lista `[2]`.

▶ 311 Diseña una función que reciba dos listas y devuelva los elementos que pertenecen a una o a otra, pero sin repetir ninguno (unión de conjuntos).

Ejemplo: si recibe las listas `[1, 2, 1]` y `[2, 3, 2, 4]`, devolverá la lista `[1, 2, 3, 4]`.

cc Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

▶ 312 Diseña una función que reciba dos listas y devuelva los elementos que pertenecen a la primera pero no a la segunda, sin repetir ninguno (diferencia de conjuntos).

Ejemplo: si recibe las listas `[1, 2, 1]` y `[2, 3, 2, 4]`, devolverá la lista `[1]`.

▶ 313 Diseña una función que, dada una lista de números, devuelva otra lista que solo incluya sus números impares.

▶ 314 Diseña una función que, dada una lista de nombres y una letra, devuelva una lista con todos los nombres que empiezan por dicha letra.

▶ 315 Diseña una función que, dada una lista de números, devuelva otra lista con solo aquellos números de la primera que son primos.

▶ 316 Diseña una función que, dada una lista de números, devuelva una lista con todos los pares de números que podemos formar con uno de la primera lista y otro de la segunda. Por ejemplo, si se suministran las listas `[1, 3, 5]` y `[2, 5]`, la lista resultante es

`[[1, 2], [1, 5], [3, 2], [3, 5], [5, 2], [5, 5]]`.

▶ 317 Diseña una función que, dada una lista de números, devuelva una lista con todos los pares de números *amigos* que podemos formar con uno de la primera lista y otro de la segunda.

6.4. El mecanismo de las llamadas a función

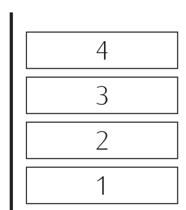
Hemos visto que desde una función podemos llamar a otra función. Desde esta última función podríamos llamar a otra, y desde esta aún a otra... Cada vez que se produce una llamada, la ejecución del programa principal o de la función «actual» queda suspendida a la espera de que finalice la llamada realizada y prosigue cuando esta finaliza. ¿Cómo recuerda Python qué funciones están «suspendidas» y en qué orden deben reanudarse?

Por otra parte, hemos visto que si una variable local a una función tiene el mismo nombre que una variable global, durante la ejecución de la función la variable local oculta a la global y su valor es inaccesible. ¿Cómo es posible que al finalizar la ejecución de una función se restaure el valor original? ¿Dónde se había almacenado este mientras la variable era invisible?

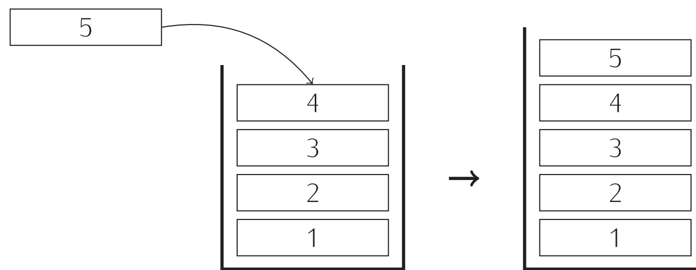
6.4.1. La pila de llamadas a función y el paso de parámetros

Python utiliza internamente una estructura especial de memoria para recordar la información asociada a cada invocación de función: la *pila de llamadas a función*. Una pila es una serie de elementos a la que solo podemos añadir y eliminar componentes por uno de sus dos extremos: el que denominamos la *cima*.

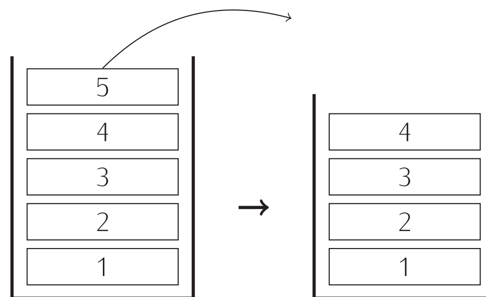
Un montón de platos, por ejemplo, es una pila: solo puedes añadir un plato poniéndolo encima de la pila (*apilar*) y solo puedes quitar el plato que está encima (*desapilar*). Aquí tienes una representación gráfica de una pila con cuatro elementos (cada uno de ellos es un número entero).



Solo podemos añadir nuevos elementos (apilar) por el extremo superior:



Y solo podemos eliminar el elemento de la cima (desapilar):



Cada activación de una función apila un nuevo componente en la pila de llamadas a función. Dicho componente, que recibe el nombre de *trama de activación*, es una zona de memoria en la que Python dispondrá espacio para los punteros asociados a parámetros, variables locales y otra información que se ha de recordar, como el punto exacto desde el que se efectuó la llamada a la función. Cuando iniciamos la ejecución de un programa, Python reserva una trama especial para las variables globales, así que empezamos con un elemento en la pila. Estudiemos un ejemplo: una ejecución particular del programa *area_y_angulo.py* que reproducimos aquí:

area_y_angulo.py
```
from math import sqrt, asin, pi

def área_triángulo(a, b, c):
    s = (a + b + c) / 2
    return sqrt(s * (s-a) * (s-b) * (s-c))

def ángulo_alfa(a, b, c):
    s = área_triángulo(a, b, c)
    return 180 / pi * asin(2 * s / (b*c))

def menú():
    opción = 0
    while opción != 1 and opción != 2:
        print('1) Calcular área del triángulo')
        print('2) Calcular ángulo opuesto al primer lado')
        opción = int(input('Escoge opción: '))
    return opción

lado1 = float(input('Dame lado a: '))
lado2 = float(input('Dame lado b: '))
lado3 = float(input('Dame lado c: '))

s = menú()

if s == 1:
    resultado = área_triángulo(lado1, lado2, lado3)
else:
    resultado = ángulo_alfa(lado1, lado2, lado3)

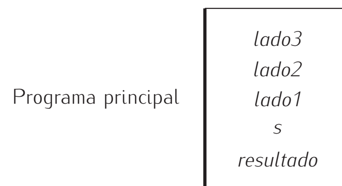
```

```
30 print('Escogiste␣la␣opción', s)
31 print('El␣resultado␣es:', resultado)
```

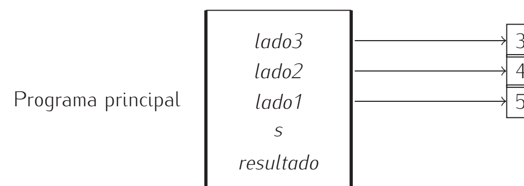
Aquí tienes un pantallazo con el resultado de dicha ejecución:

```
Dame lado a: 5↵
Dame lado b: 4↵
Dame lado c: 3↵
1) Calcular área del triángulo
2) Calcular ángulo opuesto al primer lado
Escoge opción: 2↵
Escogiste la opción 2
El resultado es: 90.0
```

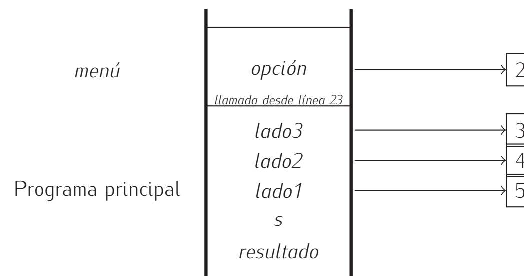
Cuando el programa arranca, Python prepara en la pila el espacio necesario para las variables globales:



El usuario introduce a continuación el valor de *lado1*, *lado2* y *lado3*. La memoria queda así:



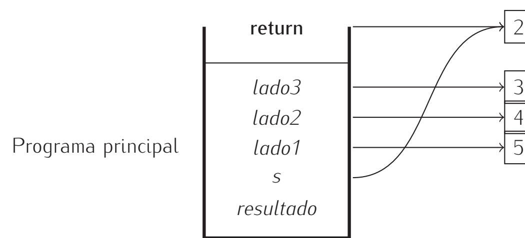
Se produce entonces la llamada a la función *menú*. Python crea una trama de activación para la llamada y la dispone en la cima de la pila. En dicha trama se almacena el valor de *opción* y el punto desde el que se efectuó la llamada a *menú*. Aquí tienes una representación de la pila cuando el usuario acaba de introducir por teclado la opción seleccionada:



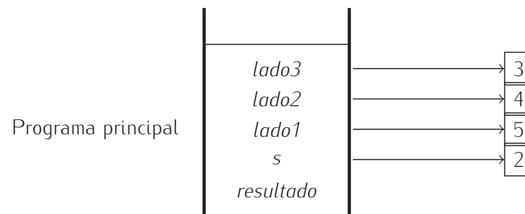
¿Qué ocurre cuando finaliza la ejecución de la función *menú*? Ya no hace falta la trama de activación, así que se desapila, es decir, se elimina. Momentáneamente, no obstante, se mantiene una referencia al objeto devuelto, en este caso, el contenido de la variable *opción*. Python recuerda en qué línea del programa principal debe continuar (línea 23) porque se había memorizado en la trama de activación. La línea 23 dice:

```
s = menú()
```

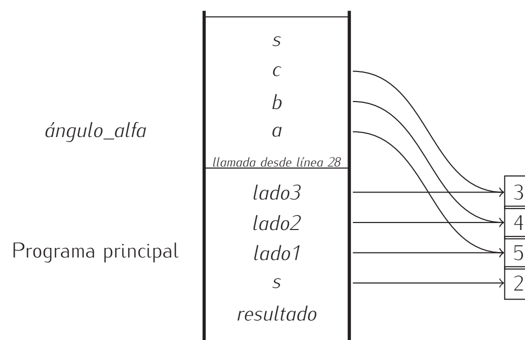
así que la referencia devuelta por *menú* con la sentencia **return** es apuntada ahora por la variable *s*:



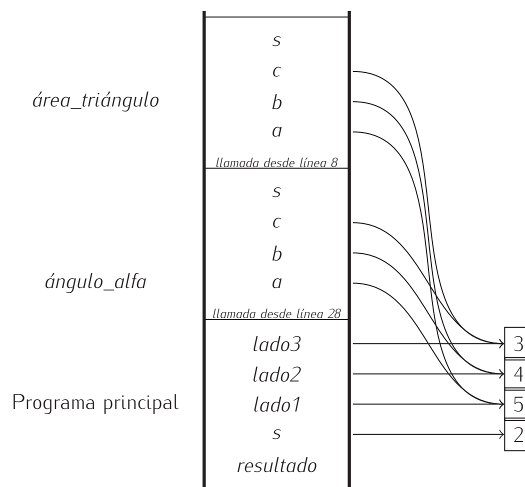
Y ahora que ha desaparecido completamente la trama de activación de *menú*, podemos reorganizar gráficamente los objetos apuntados por cada variable:



La ejecución prosigue y, en la línea 28, se produce una llamada a la función *ángulo_alfa*. Se crea entonces una nueva trama de activación en la cima de la pila con espacio para los punteros de los tres parámetros y el de la variable local *s*. A continuación, cada parámetro apunta al correspondiente valor: el parámetro *a* apunta adonde apunta *lado1*, el parámetro *b* adonde *lado2* y el parámetro *c* adonde *lado3*. Esta acción se denomina *paso de parámetros*.

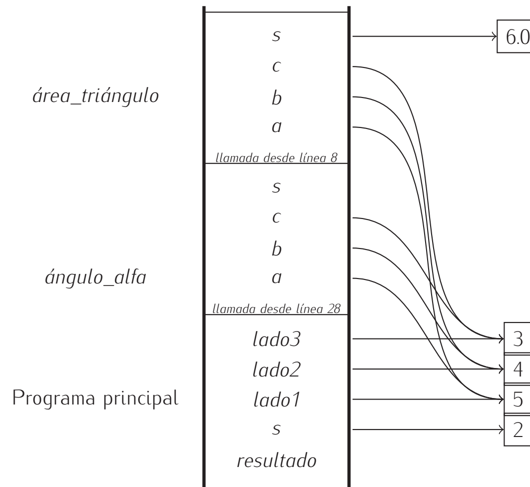


Desde el cuerpo de la función *ángulo_alfa* se llama a la función *área_triángulo*, así que se crea una nueva trama de activación. Fíjate en que los identificadores de los parámetros y las variables locales de las dos tramas superiores tienen los mismos nombres, pero residen en espacios de memoria diferentes. En esta nueva imagen puedes ver el estado de la pila en el instante preciso en que se efectúa la llamada a *área_triángulo* y se ha producido el paso de parámetros:

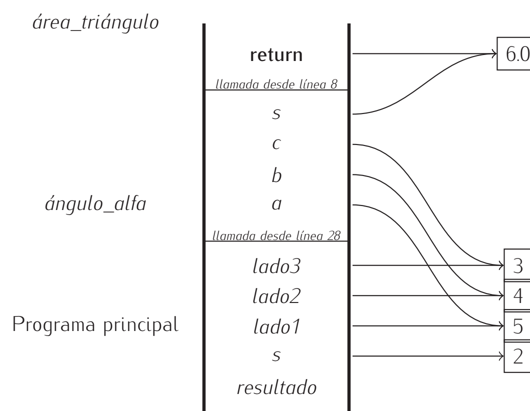


Como puedes comprobar, los parámetros *a*, *b* y *c* de *área_triángulo* apuntan al mismo lugar que los parámetros del mismo nombre de *ángulo_alfa*.

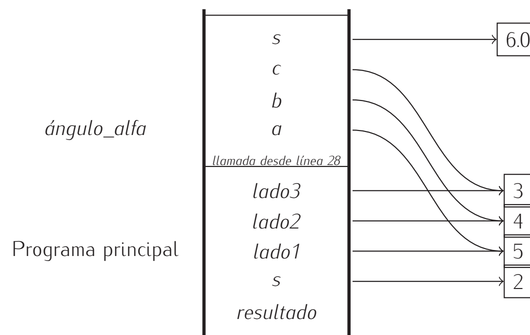
Cuando *área_triángulo* ejecuta su primera línea, la variable local *s* recibe el valor 6.0:



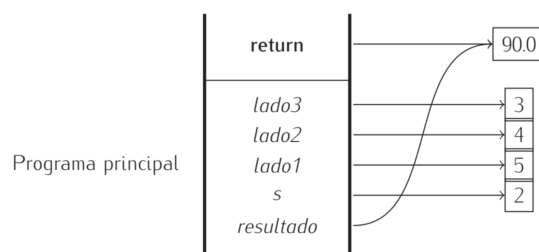
La ejecución de *área_triángulo* finaliza devolviendo el valor del área, que resulta ser 6.0. La variable *s* local a *ángulo_alfa* apunta a dicho valor, pues hay una asignación al resultado de la función en la línea 8:



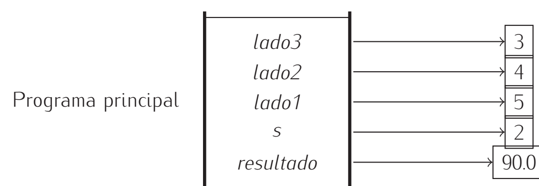
Nuevamente podemos simplificar la figura así:



Y, ahora, una vez finaliza la ejecución de *ángulo_alfa*, el valor devuelto (90.0) se almacena en la variable global *resultado*:



El estado final de la pila es, pues, este:



Observa que la variable *s* de la trama de activación del programa principal siempre ha valido 2, aunque las variables locales del mismo nombre han almacenado diferentes valores a lo largo de la ejecución del programa.

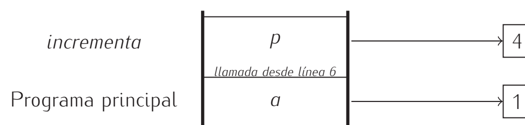
6.4.2. Paso del resultado de expresiones como argumentos

Hemos visto que el paso de parámetros comporta que el parámetro apunte a cierto lugar de la memoria. Cuando el argumento es una variable, es fácil entender qué ocurre: tanto el parámetro como la variable apuntan al mismo lugar. Pero, ¿qué ocurre si pasamos una expresión como argumento? Veamos un ejemplo:

```
parametros.py
1 def incrementa(p):
2     p = p + 1
3     return p
4
5 a = 1
6 a = incrementa(2+2)
7 print('a:', a)
```

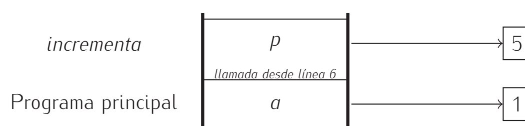
Observa que no hemos pasado a *incrementa* una variable, sino el valor 4 (resultado de evaluar la expresión 2+2).

He aquí el estado de la memoria en el preciso instante en el que se produce el paso de parámetros:



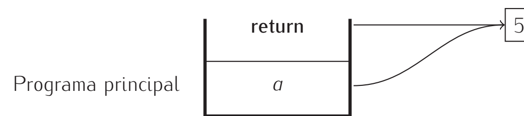
El parámetro *p* apunta a una *nueva* zona de memoria que contiene el resultado de evaluar la expresión.

La operación de incremento de la línea 2 hace que *p* pase a valer 5:



y ese es el valor devuelto en la línea 3.

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93



Así pues, la variable global *a* recibe el valor devuelto y es este el que se muestra por pantalla:

```
a: 5
```

6.4.3. Más sobre el paso de parámetros

Hemos visto que el paso de parámetros comporta que cada parámetro apunte a un lugar de la memoria y que este puede estar ya apuntado por una variable o parámetro perteneciente al ámbito desde el que se produce la llamada. ¿Qué ocurre si el parámetro es modificado dentro de la función? ¿Se modificará igualmente la variable o parámetro del ámbito desde el que se produce la llamada? Depende. Estudiemos unos cuantos ejemplos.

Para empezar, uno bastante sencillo:

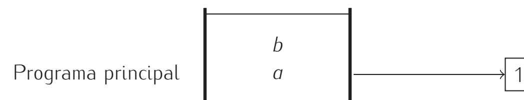
parametros.py
```
1 def incrementa(p):
2     p = p + 1
3     return p
4
5 a = 1
6 b = incrementa(a)
7
8 print('a:', a)
9 print('b:', b)
```

Veamos qué sale por pantalla al ejecutarlo:

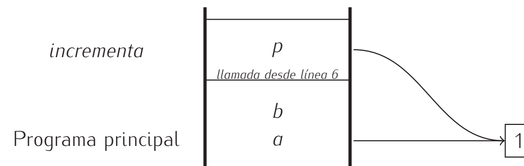
```
a: 1
b: 2
```

Puede que esperaras que tanto *a* como *b* tuvieran el mismo valor al final: a fin de cuentas la llamada a *incrementa* en la línea 6 hizo que el parámetro *p* apuntara al mismo lugar que *a* y esa función incrementa el valor de *p* en una unidad (línea 2). ¿No debería, pues, haberse modificado el valor de *a*? No.

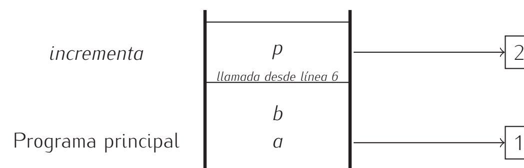
Veamos qué ocurre paso a paso. Inicialmente tenemos en la pila la reserva de memoria para las variables *a* y *b*. Tras ejecutar la línea 5, *a* tiene por valor el entero 1:



Cuando llamamos a *incrementa* el parámetro *p* recibe *una referencia al valor apuntado por a*. Así pues, tanto *a* como *p* apuntan al mismo lugar y valen 1:



El resultado de ejecutar la línea 2 ¡hace que *p* apunte a *una nueva zona de memoria* en la que se guarda el valor 2!

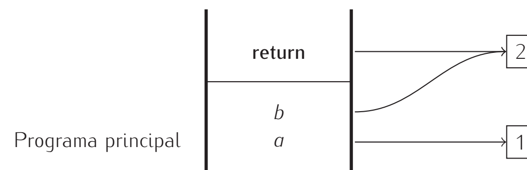


¿Por qué? Recuerda cómo procede Python ante una asignación:

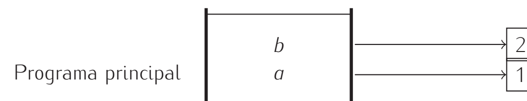
- en primer lugar se evalúa la expresión a mano derecha del igual,
- y a continuación se hace que la parte izquierda del igual apunte al resultado.

La evaluación de una expresión proporciona una referencia a la zona de memoria que alberga el resultado. Así pues, la asignación tiene un efecto sobre la referencia de *p*, no sobre el contenido de la zona de memoria apuntada por *p*. Cuando Python ha evaluado la parte derecha de la asignación de la línea 2, ha sumado al valor 1 apuntado por *p* el valor 1 que aparece explícitamente. El resultado es 2, así que Python ha reservado una nueva celda de memoria con dicho valor. Finalmente, se ha asignado a *p* el resultado de la expresión, es decir, se ha hecho que *p* apunte a la celda de memoria con el resultado.

Sigamos con la ejecución de la llamada a la función. Al finalizar esta, la referencia de *p* se devuelve y, en la línea 6, se asigna a *b*.



Resultado: *b* vale lo que valía *p* al final de la llamada y *a* no ve modificado su valor:



▶ 318 ¿Qué aparecerá por pantalla al ejecutar este programa?

parametros.py
```
def incrementa(a):
    a = a + 1
    return a

a = 1
b = incrementa(a)

print('a:', a)
print('b:', b)
```

Hazte un dibujo del estado de la pila de llamadas paso a paso para entender bien qué está pasando al ejecutar cada sentencia.

Y ahora, la sorpresa:

paso_de_listas.py
```
def modifica(a, b):
    a.append(4)
    b = b + [4]
    return b

lista1 = [1, 2, 3]
lista2 = [1, 2, 3]

lista3 = modifica(lista1, lista2)

print(lista1)
print(lista2)
```

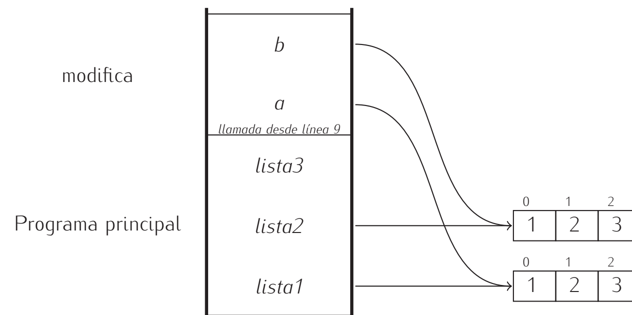
```
13 print(lista3)
```

Ejecutemos el programa:

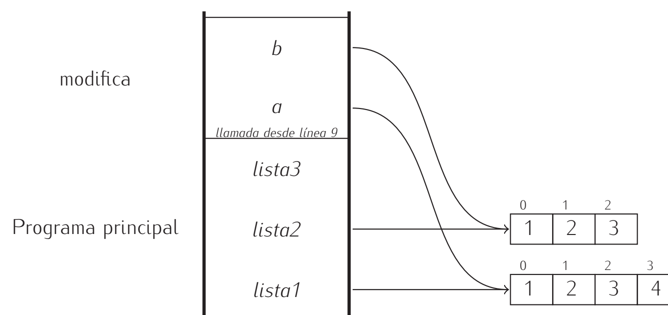
```
[1, 2, 3, 4]
[1, 2, 3]
[1, 2, 3, 4]
```

¿Qué ha ocurrido? La lista que hemos proporcionado como primer argumento se ha modificado al ejecutarse la función y la que sirvió de segundo argumento no.

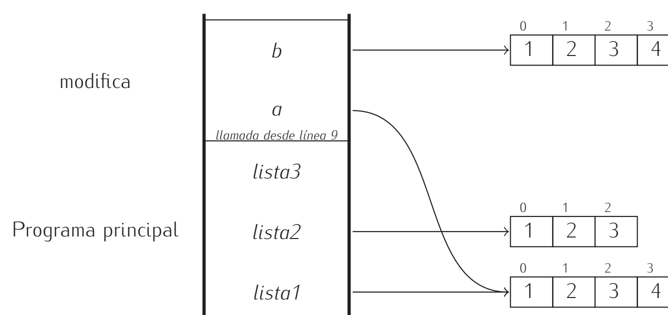
Ya deberías tener suficientes datos para averiguar qué ha ocurrido. No obstante, nos detendremos brevemente a explicarlo. Veamos en qué estado está la memoria en el momento en el que se produce el paso de parámetros en la llamada a *modifica*:



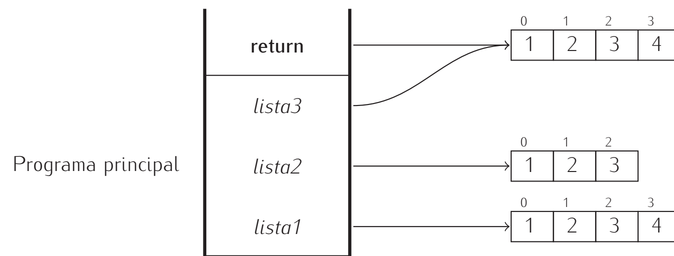
¿Qué ocurre cuando se ejecuta la línea 2? Que la lista apuntada por *a* crece por el final (con *append*) con un nuevo elemento de valor 4:



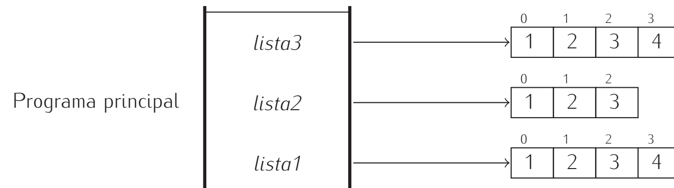
Como esa lista está apuntada tanto por el parámetro *a* como por la variable global *lista1*, ambos «sufren» el cambio y ven modificado su valor. Pasemos ahora a la línea 3: una asignación. Como siempre, Python empieza por evaluar la parte derecha de la asignación, donde se indica que se debe crear *una nueva lista* con capacidad para cuatro elementos (los valores 1, 2 y 3 que provienen de *b* y el valor 4 que aporta la lista `[4]`). Una vez creada la nueva lista, se procede a que la variable de la parte izquierda apunte a ella:



Cuando finaliza la ejecución de *modifica*, *lista3* pasa a apuntar a la lista devuelta por la función, es decir, a la lista que hasta ahora apuntaba *b*:



Y aquí tenemos el resultado final:



Recuerda, pues, que:

- La asignación puede comportar un cambio del lugar de memoria al que apunta una variable. *Si un parámetro modifica su valor mediante una asignación, (probablemente) obtendrá una nueva zona de memoria y perderá toda relación con el argumento del que tomó valor al efectuar el paso de parámetros.*
- *Operaciones como append,* **del** *o la asignación a elementos indexados de listas modifican la propia lista, por lo que los cambios afectan tanto al parámetro como al argumento.*

Con las cadenas ocurre algo similar a lo estudiado con las listas, solo que las cadenas son inmutables y no pueden sufrir cambio alguno mediante operaciones como *append*, **del** o asignación directa a elementos de la cadena. De hecho, ninguna de esas operaciones es válida sobre una cadena.

▶ 319 ¿Qué mostrará por pantalla el siguiente programa al ejecutarse?

ejercicio_parametros.py
```
def modifica(a, b):
    for elemento in b:
        a.append(elemento)
    b = b + [4]
    a[-1] = 100
    del b[0]
    return b[:]

lista1 = [1, 2, 3]
lista2 = [1, 2, 3]

lista3 = modifica(lista1, lista2)

print(lista1)
print(lista2)
print(lista3)
```

▶ 320 ¿Qué muestra por pantalla este programa al ser ejecutado?

ejercicio_parametros.py
```
def modifica_parámetros(x, y):
    x = 1
    y[0] = 1
```

```

a = 0
b = [0, 1, 2]
modifica_parámetros(a, b)

print(a)
print(b)
```

▶ 321 ¿Qué muestra por pantalla este programa al ser ejecutado?

```
ejercicio_parametros.py
def modifica_parámetros(x, y):
    x = 1
    y.append(3)
    y = y + [4]
    y[0] = 10

a = 0
b = [0, 1, 2]
modifica_parámetros(a, b)
print(a)
print(b)
```

▶ 322 Utiliza las funciones desarrolladas en el ejercicio 295 y diseña nuevas funciones para construir un programa que presente el siguiente menú y permita ejecutar las acciones correspondientes a cada opción:

```
1) Añadir estudiante y calificación
2) Mostrar lista de estudiantes con sus calificaciones
3) Calcular la media de las calificaciones
4) Calcular el número de aprobados
5) Mostrar los estudiantes con mejor calificación
6) Mostrar los estudiantes con calificación superior a la media
7) Consultar la nota de un estudiante determinado
8) FINALIZAR EJECUCIÓN DEL PROGRAMA
```

Ahora que sabemos que dentro de una función podemos modificar listas, vamos a diseñar una función que invierta una lista. ¡Ojo!: no una función que, dada una lista, *devuelva* otra que sea la inversa de la primera, sino un procedimiento (recuerda: una función que no devuelve nada) que, dada una lista, *la modifique* invirtiéndola.

El aspecto de una primera versión podría ser este:

```
inversion.py
def invierte(lista):
    for i in range(len(lista)):
        intercambiar los elementos lista[i] y lista[len(lista)-1-i]
```

Intercambiaremos los dos elementos usando una variable auxiliar:

```
inversion.py
def invierte(lista):
    for i in range(len(lista)):
        c = lista[i]
        lista[i] = lista[len(lista)-1-i]
        lista[len(lista)-1-i] = c

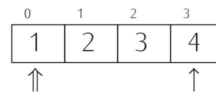
a = [1, 2, 3, 4]
invierte(a)
print(a)
```

No funciona. Parece que no la haya modificado. En realidad sí que lo ha hecho, pero mal. Estudiemos paso a paso qué ha ocurrido:

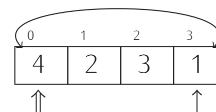
1) Al llamar a la función, el parámetro *lista* «apunta» (hace referencia) a la misma zona de memoria que la variable *a*.

2) El bucle que empieza en la línea 2 va de 0 a 3 (pues la longitud de *lista* es 4). La variable local *i* tomará los valores 0, 1, 2 y 3.

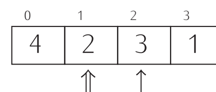
   1) Cuando *i* vale 0, el método considera los elementos *lista*[0] y *lista*[3]:



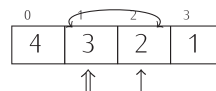
   La variable local *c* toma el valor 1 (que es el contenido de *lista*[0]), a continuación *lista*[0] toma el valor de *lista*[3] y, finalmente, *lista*[3] toma el valor de *c*. El resultado es que se intercambian los elementos *lista*[0] y *lista*[3]:



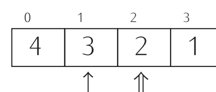
   2) Ahora *i* vale 1, así que se consideran los elementos *lista*[1] y *lista*[2]:



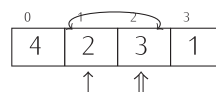
   Los dos elementos se intercambian y la lista queda así:



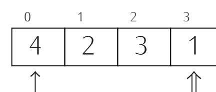
   3) Ahora *i* vale 2, así que se consideran los elementos *lista*[2] y *lista*[1]:



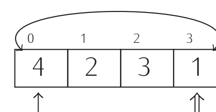
   Tras el intercambio, la lista pasa a ser:



   4) Y, finalmente, *i* vale 3.



   Se intercambian los valores de las celdas *lista*[3] y *lista*[0]:



Fíjate en que al final de la segunda iteración del bucle la lista estaba correctamente invertida. Lo que ha ocurrido es que hemos seguido iterando y ¡hemos vuelto a invertir una lista que ya estaba invertida, dejándola como estaba al principio! Ya está claro cómo actuar: iterando la mitad de las veces. Vamos allá:

```
inversion.py
def invierte(lista):
    for i in range(len(lista)//2):
        c = lista[i]
        lista[i] = lista[len(lista)-1-i]
        lista[len(lista)-1-i] = c

a = [1, 2, 3, 4]
invierte(a)
print(a)
```

Ahora sí. Si ejecutamos el programa obtenemos:

```
[4, 3, 2, 1]
```

▶ 323 ¿Qué ocurre con el elemento central de la lista cuando la lista tiene un número impar de elementos? ¿Nuestra función invierte correctamente la lista?

▶ 324 Un aprendiz sugiere esta otra solución. ¿Funciona?

```
inversion.py
def invierte(lista):
    for i in range(len(lista)//2):
        c = lista[i]
        lista[i] = lista[-i-1]
        lista[-i-1] = c
```

▶ 325 ¿Qué muestra por pantalla este programa al ser ejecutado?

```
abslista.py
def abs_lista(lista):
    for i in range(len(lista)):
        lista[i] = abs(lista[i])

milista = [1, -1, 2, -3, -2, 0]
abs_lista(milista)
print(milista)
```

▶ 326 ¿Qué mostrará por pantalla el siguiente programa al ejecutarse?

```
intercambio.py
def intento_de_intercambio(a, b):
    aux = a
    a = b
    b = aux

lista1 = [1, 2]
lista2 = [3, 4]

intento_de_intercambio(lista1, lista2)

print(lista1)
print(lista2)
```

▶ 327 Diseña un procedimiento que, dada una lista de números, la modifique para que solo sobrevivan a la llamada aquellos números que son perfectos.

▶ 328 Diseña una función *duplica* que reciba una lista de números y la modifique duplicando el valor de cada uno de sus elementos. (Ejemplo: la lista `[1, 2, 3]` se convertirá en la lista `[2, 4, 6]`).

▶ 329 Diseña una función *duplica_copia* que reciba una lista de números y devuelva *otra* lista en la que cada elemento sea el doble del que tiene el mismo índice en la lista original. La lista original *no debe sufrir ninguna modificación* tras la llamada a *duplica_copia*.

▶ 330 Diseña una función que reciba una lista y devuelva otra lista cuyo contenido sea el resultado de concatenar la lista original consigo misma. La lista original no debe modificarse.

▶ 331 Diseña una función que reciba una lista y devuelva otra lista cuyo contenido sea la lista original, pero con sus componentes en orden inverso. La lista original no debe modificarse.

▶ 332 Diseña una función que reciba una lista y devuelva una lista cuyo contenido sea la lista original concatenada con una versión invertida de ella misma. La lista original no debe modificarse.

▶ 333 Diseña un procedimiento que reciba una lista y ordene sus elementos de menor a mayor.

▶ 334 Diseña una función que reciba una lista y devuelva una copia de la lista con sus elementos ordenados de menor a mayor. La lista original no debe modificarse.

▶ 335 Diseña una función que reciba una matriz y, si es cuadrada (es decir, tiene igual número de filas que de columnas), devuelva la suma de todos los componentes dispuestos en la diagonal principal (es decir, todos los elementos de la forma $A_{i,i}$). Si la matriz no es cuadrada, la función devolverá **None**.

▶ 336 Guardamos en una matriz de $m \times n$ elementos la calificación obtenida por m estudiantes (a los que conocemos por su número de lista) en la evaluación de n ejercicios entregados semanalmente (cuando un ejercicio no se ha entregado, la calificación es -1).

Diseña funciones y procedimientos que efectúen los siguiente cálculos:

- Dado el número de un alumno, devolver el número de ejercicios entregados.
- Dado el número de un alumno, devolver la media sobre los ejercicios entregados.
- Dado el número de un alumno, devolver la media sobre los ejercicios entregados si los entregó todos; en caso contrario, la media es 0.
- Devolver el número de todos los alumnos que han entregado todos los ejercicios y tienen una media superior a 3,5 puntos.
- Dado el número de un ejercicio, devolver el número de estudiantes que lo han presentado.
- Dado el número de un ejercicio, devolver la nota media obtenida por los estudiantes que lo presentaron.
- Dado el número de un ejercicio, devolver la nota más alta obtenida.
- Dado el número de un ejercicio, devolver la nota más baja obtenida.
- Devolver el número de abandonos en función de la semana. Consideramos que un alumno abandonó en la semana s si no ha entregado ningún ejercicio desde entonces. Este procedimiento mostrará en pantalla el número de abandonos para cada semana (si un alumno no ha entregado nunca ningún ejercicio, abandonó en la «semana cero»).

6.4.4. Acceso a variables globales desde funciones

Por lo dicho hasta ahora podrías pensar que en el cuerpo de una función solo pueden utilizarse variables locales. No es cierto. Dentro de una función también puedes consultar y modificar variables globales. Eso sí, deberás «avisar» a Python de que una variable usada en el cuerpo de una función es global *antes* de usarla. Lo veremos con un ejemplo.

Vamos a diseñar un programa que gestiona una de las funciones de un cajero automático que puede entregar cantidades que son múltiplo de 10 €. En cada momento, el cajero tiene un número determinado de billetes de 50, 20 y 10 €. Utilizaremos una variable para cada tipo de billete y en ella indicaremos cuántos billetes de ese tipo nos quedan en el cajero. Cuando un cliente pida sacar una cantidad determinada de dinero, mostraremos por pantalla cuántos billetes de cada tipo le damos. Intentaremos darle siempre la menor cantidad de billetes posible. Si no es posible darle el dinero (porque no tenemos suficiente dinero en el cajero o porque la cantidad solicitada no puede darse con una combinación válida de los billetes disponibles) informaremos al usuario.

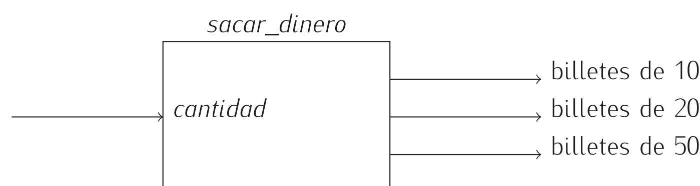
Inicialmente supondremos que el cajero está cargado con 100 billetes de cada tipo:

```
cajero.py
carga50 = 100
carga20 = 100
carga10 = 100
```

Diseñaremos ahora una función que, ante una petición de dinero, muestre por pantalla los billetes de cada tipo que se entregan. La función devolverá una lista con el número de billetes de 50, 20 y 10 € si se pudo dar el dinero, y la lista `[0, 0, 0]` en caso contrario. Intentémoslo.

```
cajero.py
carga50 = 100
carga20 = 100
carga10 = 100

def sacar_dinero(cantidad):
    de50 = cantidad // 50
    cantidad = cantidad % 50
    de20 = cantidad // 20
    cantidad = cantidad % 20
    de10 = cantidad // 10
    return [de50, de20, de10]
```



¿Entiendes las fórmulas utilizadas para calcular el número de billetes de cada tipo? Estúdialas con calma antes de seguir.

En principio, ya está. Bueno, no; hemos de restar los billetes que le damos al usuario de las variables *carga50*, *carga20* y *carga10*, pues el cajero ya no los tiene disponibles para futuras extracciones de dinero:

```
cajero.py
carga50 = 100
carga20 = 100
carga10 = 100

def sacar_dinero(cantidad):
    de50 = cantidad // 50
```

```
7      cantidad = cantidad % 50
8      de20 = cantidad // 20
9      cantidad = cantidad % 20
10     de10 = cantidad // 10
11     carga50 = carga50 - de50
12     carga20 = carga20 - de20
13     carga10 = carga10 - de10
14     return [de50, de20, de10]
```

Probemos el programa añadiendo, momentáneamente, un programa principal:

```
cajero.py
1  carga50 = 100
2  carga20 = 100
3  carga10 = 100
4
5  def sacar_dinero(cantidad):
6      de50 = cantidad // 50
7      cantidad = cantidad % 50
8      de20 = cantidad // 20
9      cantidad = cantidad % 20
10     de10 = cantidad // 10
11     carga50 = carga50 - de50
12     carga20 = carga20 - de20
13     carga10 = carga10 - de10
14     return [de50, de20, de10]
15
16 c = int(input('Cantidad␣a␣extraer:␣'))
17 print(sacar_dinero(c))
```

¿Qué ocurrirá con el acceso a *carga50*, *carga20* y *carga10*? Puede que Python las tome por variables locales, en cuyo caso, no habremos conseguido el objetivo de actualizar la cantidad de billetes disponibles de cada tipo. Lo que ocurre es peor aún: al ejecutar el programa obtenemos un error.

```
Cantidad a extraer: 70↵
Traceback (most recent call last):
  File "cajero.py", line 17, in <module>
    print(sacar_dinero(c))
  File "cajero.py", line 11, in sacar_dinero
    carga50 = carga50 - de50
UnboundLocalError: local variable 'carga50' referenced before assignment
```

El error es del tipo *UnboundLocalError* (que podemos traducir por «error de variable local no ligada») y nos indica que hubo un problema al tratar de acceder a *carga50*, pues es una variable *local* que no tiene valor asignado previamente. Pero, ¡*carga50* debería ser una variable global, no local, y además sí se le asignó un valor: en la línea 1 asignamos a *carga50* el valor 100! ¿Por qué se confunde? Python utiliza una regla simple para decidir si una variable usada en una función es local o global: si se le asigna un valor, es local; si no, es global. Las variables *carga50*, *carga20* y *carga10* aparecen en la parte izquierda de una asignación, así que Python supone que son variables locales. Y si son locales, no están inicializadas cuando se evalúa la parte derecha de la asignación. Hay una forma de evitar que Python se equivoque en situaciones como esta: declarar explícitamente que esas variables son globales. Fíjate en la línea 6:

```
cajero.py
1  carga50 = 100
2  carga20 = 100
3  carga10 = 100
4
5  def sacar_dinero(cantidad):
6      global carga50, carga20, carga10
7      de50 = cantidad // 50
```

```
    cantidad = cantidad % 50
    de20 = cantidad // 20
    cantidad = cantidad % 20
    de10 = cantidad // 10
    carga50 = carga50 - de50
    carga20 = carga20 - de20
    carga10 = carga10 - de10
    return [de50, de20, de10]

c = int(input('Cantidad a extraer: '))
print(sacar_dinero(c))
```

```
Cantidad a extraer: 70
[1, 1, 0]
```

¡Perfecto! Hagamos una prueba más:

```
Cantidad a extraer: 7000
[140, 0, 0]
```

¿No ves nada raro? ¡La función ha dicho que nos han de dar 140 billetes de 50 €, cuando solo hay 100! Hemos de refinar la función y hacer que nos dé la cantidad solicitada solo cuando dispone de suficiente efectivo:

```
cajero.py
1  carga50 = 100
2  carga20 = 100
3  carga10 = 100
4
5  def sacar_dinero(cantidad):
6      global carga50, carga20, carga10
7      if cantidad <= 50 * carga50 + 20 * carga20 + 10 * carga10:
8          de50 = cantidad // 50
9          cantidad = cantidad % 50
10         de20 = cantidad // 20
11         cantidad = cantidad % 20
12         de10 = cantidad // 10
13         carga50 = carga50 - de50
14         carga20 = carga20 - de20
15         carga10 = carga10 - de10
16         return [de50, de20, de10]
17     else:
18         return [0, 0, 0]
19
20 c = int(input('Cantidad a extraer: '))
21 print(sacar_dinero(c))
```

La línea 7 se encarga de averiguar si hay suficiente dinero en el cajero. Si no lo hay, la función finaliza inmediatamente devolviendo la lista [0, 0, 0]. ¿Funcionará ahora?

```
Cantidad a extraer: 7000
[140, 0, 0]
```

¡No! Sigue funcionando mal. ¡Claro!, hay $50 \times 100 + 20 \times 100 + 10 \times 100 = 8000$ € en el cajero y hemos pedido 7000 €. Lo que deberíamos controlar no (solo) es que haya suficiente dinero, sino que haya suficiente cantidad de billetes de cada tipo:

```
cajero.py
carga50 = 100
carga20 = 100
carga10 = 100

def sacar_dinero(cantidad):
```

```
    global carga50, carga20, carga10
    if cantidad <= 50 * carga50 + 20 * carga20 + 10 * carga10:
        de50 = cantidad // 50
        cantidad = cantidad % 50
        if de50 >= carga50: # Si no hay suficientes billetes de 50
            cantidad = cantidad + (de50 - carga50) * 50
            de50 = carga50
        de20 = cantidad // 20
        cantidad = cantidad % 20
        if de20 >= carga20: # y no hay suficientes billetes de 20
            cantidad = cantidad + (de20 - carga20) * 20
            de20 = carga20
        de10 = cantidad // 10
        cantidad = cantidad % 10
        if de10 >= carga10: # y no hay suficientes billetes de 10
            cantidad = cantidad + (de10 - carga10) * 10
            de10 = carga10
        # Si todo ha ido bien, la cantidad que resta por entregar es nula:
        if cantidad == 0:
            # Así que hacemos efectiva la extracción
            carga50 = carga50 - de50
            carga20 = carga20 - de20
            carga10 = carga10 - de10
            return [de50, de20, de10]
        else: # Y si no, devolvemos la lista con tres ceros:
           return [0, 0, 0]
    else:
        return [0, 0, 0]

c = int(input('Cantidad a extraer: '))
print(sacar_dinero(c))
```

Bueno, parece que ya tenemos la función completa. Hagamos algunas pruebas:

```
Cantidad a extraer: 130↵
[2, 1, 1]

Cantidad a extraer: 7000↵
[100, 100, 0]

Cantidad a extraer: 9000↵
[0, 0, 0]
```

¡Ahora sí!

▶ 337 Hay dos ocasiones en las que se devuelve la lista [0, 0, 0]. ¿Puedes modificar el programa para que solo se devuelva esa lista explícita desde un punto del programa?

Como ya hemos diseñado y probado la función, hagamos un último esfuerzo y acabemos el programa. Eliminamos las líneas de prueba (las dos últimas) y añadimos el siguiente código:

cajero.py
```
carga50 = 100
carga20 = 100
carga10 = 100

def sacar_dinero(cantidad):
    global carga50, carga20, carga10
    if cantidad <= 50 * carga50 + 20 * carga20 + 10 * carga10:
        de50 = cantidad // 50
        cantidad = cantidad % 50
        if de50 >= carga50: # Si no hay suficientes billetes de 50
            cantidad = cantidad + (de50 - carga50) * 50
```

```
            de50 = carga50
        de20 = cantidad // 20
        cantidad = cantidad % 20
        if de20 >= carga20: # y no hay suficientes billetes de 20
            cantidad = cantidad + (de20 - carga20) * 20
            de20 = carga20
        de10 = cantidad // 10
        cantidad = cantidad % 10
        if de10 >= carga10: # y no hay suficientes billetes de 10
            cantidad = cantidad + (de10 - carga10) * 10
            de10 = carga10
        # Si todo ha ido bien, la cantidad que resta por entregar es nula:
        if cantidad == 0:
            # Así que hacemos efectiva la extracción
            carga50 = carga50 - de50
            carga20 = carga20 - de20
            carga10 = carga10 - de10
            return [de50, de20, de10]
        else: # Y si no, devolvemos la lista con tres ceros:
           return [0, 0, 0]
    else:
        return [0, 0, 0]

# Programa principal
while 50*carga50 + 20*carga20 + 10*carga10 > 0:
    petición = int(input('Cantidad que desea sacar: '))
    [de50, de20, de10] = sacar_dinero(petición)
    if [de50, de20, de10] != [0, 0, 0]:
        if de50 > 0:
            print('Billetes de 50 euros:', de50)
        if de20 > 0:
            print('Billetes de 20 euros:', de20)
        if de10 > 0:
            print('Billetes de 10 euros:', de10)
        print('Gracias por usar el cajero.\n')
    else:
        print('Lamentamos no poder atender su petición.\n')
print('Cajero sin dinero. Avise a mantenimiento.')
```

Usemos esta versión final del programa:

```
Cantidad que desea sacar: 7000↵
Billetes de 50 euros: 100
Billetes de 20 euros: 100
Gracias por usar el cajero.
Cantidad que desea sacar: 500↵
Billetes de 10 euros: 50
Gracias por usar el cajero.
Cantidad que desea sacar: 600↵
Lamentamos no poder atender su petición.
Cantidad que desea sacar: 500↵
Billetes de 10 euros: 50
Gracias por usar el cajero.
Cajero sin dinero. Avise a mantenimiento.
```

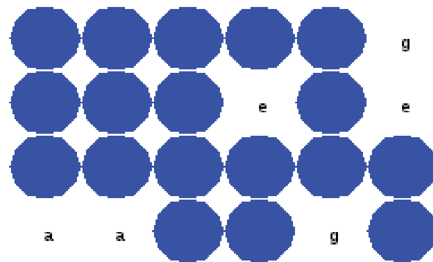
Acabaremos este apartado con una reflexión. Ten en cuenta que modificar variables globales desde una función no es una práctica de programación recomendable. La experiencia dice que solo en contadas ocasiones está justificado que una función modifique variables globales. Se dice que modificar variables globales desde una función es un *efecto secundario* de la llamada a la función. Si cada función de un programa largo modificara libremente el valor de variables globales, tu programa sería bastante ilegible y, por tanto, difícil de ampliar o corregir en el futuro.

> **Se supone que un cajero de verdad debe entregar dinero**
>
> El programa del cajero automático no parece muy útil: se limita a imprimir por pantalla el número de billetes de cada tipo que nos ha de entregar. Se supone que un cajero de verdad debe entregar dinero y no limitarse a mostrar mensajes por pantalla.
>
> Los cajeros automáticos están gobernados por un computador. Las acciones del cajero pueden controlarse por medio de funciones especiales. Estas funciones acceden a *puertos de entrada/salida* del ordenador que se comunican con los periféricos adecuados. El aparato que entrega billetes no es más que eso, un periférico más.
>
> Lo lógico sería disponer de un módulo, digamos *dispensador_de_billetes*, que nos diera acceso a las funciones que controlan el periférico. Una función podría, por ejemplo, entregar al usuario tantos billetes de cierto tipo como se indicara. Si dicha función se llamara *entrega*, en lugar de una sentencia como «`print('Billetes de 50 euros:', de50)`», realizaríamos la llamada `entrega(de50, 50)`.

6.5. Un ejemplo: Memorión

Ya es hora de hacer algo interesante con todo lo que hemos aprendido. Vamos a construir un sencillo juego solitario, Memorión, con el que aprenderemos, entre otras cosas, a manejar el ratón. Memorión se juega sobre un tablero de 4 filas y 6 columnas. Cada celda del tablero contiene un símbolo (una letra), pero no es visible porque está tapado por una baldosa. De cada símbolo hay dos ejemplares (dos «a»», dos «b», etc.) y hemos de emparejarlos. Una jugada consiste en levantar dos baldosas para ver las letras que hay bajo ellas. Primero se levanta una pulsando en la baldosa, con lo que se destapa el símbolo que oculta, y después se destapa la otra y se muestra brevemente su símbolo oculto correspondiente. Si las letras descubiertas son iguales, hemos conseguido un emparejamiento y las baldosas se retiran del tablero, dejando las letras definitivamente al descubierto. Si, por el contrario, las letras son diferentes, se tapan de nuevo. Encontrar sus respectivas parejas dependerá entonces de la memoria del jugador. El objetivo es emparejar todas las letras en el menor número de jugadas.

Esta figura te muestra una partida de Memorión ya empezada:



¿Por dónde empezamos a escribir el programa? Pensemos en qué información necesitaremos. Por una parte, necesitaremos una matriz con 4×6 celdas para almacenar las letras. Por otra parte, otra matriz «paralela» que nos diga si la casilla ya está descubierta o permanece cerrada. Nos vendrá bien disponer de una rutina que construya una matriz con la dimensión que especifiquemos, pues la usaremos tanto para construir la matriz de letras como la matriz de baldosas:

memorion.py
```
def crea_matriz(filas, columnas):
    matriz = []
    for i in range(filas):
        matriz.append([None] * columnas)
    return matriz
```

En el programa principal usaremos esta rutina así:

memorion.py
```
...
```

```
# Programa principal
filas = 4
columnas = 6
símbolo = crea_matriz(filas, columnas)
tablero = crea_matriz(filas, columnas)
```

Para inicializar el tablero con todas las casillas cerradas bastará con asignar un valor que represente ese estado en todas las celdas de su matriz. En principio parece que solo haya dos estados posibles, pero si lo piensas bien, verás que en realidad hay tres estados diferentes: la celda está tapada, la celda está destapada… y la celda está temporalmente destapada. El tercer estado corresponde al momento en que estamos viendo el símbolo que esconde una baldosa por haber hecho clic sobre ella, pero sin haber logrado aún un emparejamiento con ella. Usaremos tres valores para representar estos tres estados. En aras de la legibilidad del programa, cada valor se almacenará en una variable con un identificador explicativo:

```
memorion.py
CeldaCerrada = 0
CeldaAbierta = 1
CeldaTemporalmenteAbierta = 2

def inicializa_tablero(tablero):
    for i in range(filas):
        for j in range(columnas):
            tablero[i][j] = CeldaCerrada
```

Nuestro primer problema importante es inicializar la matriz de letras al azar. ¿Cómo podemos resolverlo? Te sugerimos que consideres estas estrategias:

- Como vamos a ubicar 12 letras diferentes (dos ejemplares de cada), un bucle va recorriendo los caracteres de la cadena `'abcdefghijkl'`. Para cada letra, elegimos dos pares de coordenadas al azar (ahora veremos cómo). Imagina que decidimos que la letra `'f'` va a las posiciones (i, j) y (i', j'), donde i e i' son números de fila y j y j' son números de columna. Hemos de asegurarnos de que las casillas (i, j) e (i', j') son diferentes y no están ya ocupadas con otras letras. (Ten en cuenta que hemos generado esos pares de números al azar, así que pueden caer en cualquier sitio y este no tiene por qué estar libre). Mientras generemos un par de coordenadas que corresponden a una casilla ocupada, repetiremos la tirada.

```
memorion.py
from random import randrange
...
def rellena_símbolos(símbolo): # Primera versión.
    for carácter in 'abcdefghijkl':
        for ejemplar in range(2):
            ocupado = True
            while ocupado:
                [i, j] = [randrange(len(símbolo)), randrange(len(símbolo[0]))]
                if símbolo[i][j] == None:
                    ocupado = False
            símbolo[i][j] = carácter
```

Para generar el número de fila y columna usamos *randrange*, que devuelve un valor aleatorio y entero mayor o igual que 0 y menor que el valor del argumento que se le proporciona. Resulta útil conocer bien las librerías estándar de Python.

En principio, no ha sido demasiado complicado diseñar esta función, pero el método por el que elegimos casillas al azar presenta un serio problema: como genera coordenadas al azar hasta dar con una libre, ¿qué ocurrirá, probablemente, cuando queden muy pocas libres? Imagina que seguimos esta estrategia en un tablero de 1000 por 1000 casillas. Cuando

solo queden dos libres, es probable que tengamos que generar muchísimas «tiradas de dado» hasta dar con una casilla libre: la probabilidad de que demos con una de ellas es de una contra medio millón. Eso significa que, en promedio, hará falta echar medio millón de veces los dados para encontrar una casilla libre. Ineficiente a más no poder.

- Creamos una lista con todos los pares de coordenadas posibles, o sea, una lista de listas: `[[0,0], [0,1], [0,2], ..., [3, 5]]`. A continuación, desordenamos la lista. ¿Cómo? Escogiendo muchas veces (por ejemplo, mil veces) un par de elementos de la lista e intercambiándolos. Una vez desordenada la lista, la usamos para asignar los caracteres:

memorion.py
```
def rellena_símbolos(símbolo): # Segunda versión.
    lista = []
    for i in range(len(símbolo)):
        for j in range(len(símbolo[0])):
            lista.append( [i, j] )

    for i in range(1000):
        [i, j] = [randrange(len(símbolo)), randrange(len(símbolo[0]))]
        aux = lista[i]
        lista[i] = lista[j]
        lista[j] = aux

    i = 0
    for coords in lista:
        símbolo[coords[0]][coords[1]] = 'abcdefghijkl'[i//2]
    i += 1
```

Complicado, ¿verdad? No solo es complicado; además, presenta un serio inconveniente: un elevado (y gratuito) consumo de memoria. Imagina que la matriz tiene dimensión 1000×1000: hemos de construir una lista con un millón de elementos y barajarlos (para lo que necesitaremos bastante más que 1000 intercambios). Una lista tan grande ocupa mucha memoria. La siguiente solución es igual de efectiva y no consume tanta memoria.

- Ponemos las letras ordenadamente en la matriz. Después, intercambiamos mil veces un par de casillas escogidas al azar:

memorion.py
```
def rellena_símbolos(símbolo): # Tercera versión.
    numsímbolo = 0
    for i in range(len(símbolo)):
        for j in range(len(símbolo[0])):
            símbolo[i][j] = chr(ord('a') + numsímbolo // 2)
            numsímbolo += 1

    for i in range(1000):
        [f1, c1] = [randrange(len(símbolo)), randrange(len(símbolo[0]))]
        [f2, c2] = [randrange(len(símbolo)), randrange(len(símbolo[0]))]
        tmp = símbolo[f1][c1]
        símbolo[f1][c1] = símbolo[f2][c2]
        símbolo[f2][c2] = tmp
```

Estudia con cuidado esta función. Es la que vamos a usar en nuestro programa.

Bueno. Ya le hemos dedicado bastante tiempo a la inicialización de la matriz de símbolos. Ahora vamos a dibujar en pantalla su contenido. Necesitamos inicializar la pantalla y la tortuga con la que dibujaremos todo.

```
memorion.py
...

# Programa principal
filas = 4
columnas = 6

pantalla = Screen()
pantalla.setup(columnas*50, filas*50)
pantalla.screensize(columnas*50, filas*50)
pantalla.setworldcoordinates(-.5, -.5, columnas+.5, filas+.5)
pantalla.delay(0)
tortuga = Turtle()
tortuga.hideturtle()
símbolo = crea_matriz(filas, columnas)
tablero = crea_matriz(filas, columnas)

inicializa_tablero(tablero)
rellena_símbolos(símbolo)
dibuja_tablero(tablero, símbolo)
```

El procedimiento *dibuja_tablero* recibe las matrices con el estado de las celdas y los símbolos y hace algo muy sencillo: recurre a otra función para dibujar cada una de las celdas en el estado en el que se encuentre:

```
memorion.py
def dibuja_tablero(tablero, símbolo):
    for i in range(len(símbolo)):
        for j in range(len(símbolo[0])):
            dibuja_celda(tablero, símbolo, i, j)
```

La función *dibuja_celda* es la responsable de dibujar el contenido de cada celda individual:

```
memorion.py
def dibuja_celda(baldosa, símbolo, i, j):
    global tortuga
    tortuga.penup()
    tortuga.goto(j+.5, i)
    tortuga.begin_fill()
    if baldosa[i][j] == CeldaCerrada:
        tortuga.fillcolor('blue')
        tortuga.circle(.5)
    elif baldosa[i][j] == CeldaAbierta:
        tortuga.fillcolor('white')
        tortuga.circle(.5)
        tortuga.goto(j+.5, i+.25)
        tortuga.write(símbolo[i][j])
    else:
        tortuga.fillcolor('yellow')
        tortuga.circle(.5)
        tortuga.goto(j+.5, i+.25)
        tortuga.write(símbolo[i][j])
    tortuga.end_fill()
    tortuga.pendown()
```

Hemos recurrido a métodos de la tortuga que aún no conoces. Veamos qué hacen:

- *tortuga.begin_fill*(): Inicia un polígono relleno.
- *tortuga.end_fill*(): Rellena el polígono iniciado con *tortuga.begin_fill*().
- *tortuga.fill_color*(*color*): Hace que el color de relleno sea el que se especifica con una cadena como argumento.

Pongamos en un único fichero todo lo que hemos hecho de momento.

```
memorion.py
from random import randrange
from turtle import Screen, Turtle

CeldaCerrada = 0
CeldaAbierta = 1
CeldaTemporalmenteAbierta = 2

def crea_matriz(filas, columnas):
    matriz = []
    for i in range(filas):
        matriz.append([None] * columnas)
    return matriz

def rellena_símbolos(símbolo): # Tercera versión.
    numsímbolo = 0
    for i in range(len(símbolo)):
        for j in range(len(símbolo[0])):
            símbolo[i][j] = chr(ord('a') + numsímbolo // 2)
            numsímbolo += 1

    for i in range(1000):
        [f1, c1] = [randrange(len(símbolo)), randrange(len(símbolo[0]))]
        [f2, c2] = [randrange(len(símbolo)), randrange(len(símbolo[0]))]
        tmp = símbolo[f1][c1]
        símbolo[f1][c1] = símbolo[f2][c2]
        símbolo[f2][c2] = tmp

def inicializa_tablero(tablero):
    for i in range(len(tablero)):
        for j in range(len(tablero[0])):
            tablero[i][j] = CeldaCerrada

def dibuja_tablero(tablero, símbolo):
    for i in range(len(símbolo)):
        for j in range(len(símbolo[0])):
            dibuja_celda(tablero, símbolo, i, j)

def dibuja_celda(baldosa, símbolo, i, j):
    global tortuga
    tortuga.penup()
    tortuga.goto(j+.5, i)
    tortuga.begin_fill()
    if baldosa[i][j] == CeldaCerrada:
        tortuga.fillcolor('blue')
        tortuga.circle(.5)
    elif baldosa[i][j] == CeldaAbierta:
        tortuga.fillcolor('white')
        tortuga.circle(.5)
        tortuga.goto(j+.5, i+.25)
        tortuga.write(símbolo[i][j])
    else:
        tortuga.fillcolor('yellow')
        tortuga.circle(.5)
        tortuga.goto(j+.5, i+.25)
        tortuga.write(símbolo[i][j])
    tortuga.end_fill()
    tortuga.pendown()

# Programa principal
```

```
filas = 4
columnas = 6

pantalla = Screen()
pantalla.setup(columnas*50, filas*50)
pantalla.screensize(columnas*50, filas*50)
pantalla.setworldcoordinates(-.5, -.5, columnas+.5, filas+.5)
pantalla.delay(0)
tortuga = Turtle()
tortuga.hideturtle()
símbolo = crea_matriz(filas, columnas)
tablero = crea_matriz(filas, columnas)

inicializa_tablero(tablero)
rellena_símbolos(símbolo)
dibuja_tablero(tablero, símbolo)

pantalla.exitonclick()
```

Ejecuta el programa y verás en pantalla el tablero de juego como una simple matriz de círculos azules. Vamos ya a por lo difícil: la interacción con el ratón. El modo de trabajar es un tanto especial. Definiremos una función a la que el sistema llamará automáticamente cada vez que se produzca un clic de ratón. La función debe tener dos parámetros: las coordenadas del punto en el que se ha producido el clic de ratón (expresada en las coordenadas de pantalla que hemos definido). Hagamos, de momento, que la función escriba las coordenadas de la celda seleccionada:

memorion.py

```
def clic(x, y):
    [j, i] = [int(x), int(y)]
    print('Clic en fila {0} y columna {1}'.format(i, j))
```

Tendremos que «conectar» la función con el evento de ratón a través del método *pantalla.onclick*. El método *pantalla.onclick* admite como argumento una función: la que queremos que se llame automáticamente cada vez que se pulsa el botón del ratón. El cableado del evento a la función que hemos definido tiene lugar en el programa principal:

memorion.py

```
...

# Programa principal
filas = 4
columnas = 6

pantalla = Screen()
pantalla.setup(columnas*50, filas*50)
pantalla.screensize(columnas*50, filas*50)
pantalla.setworldcoordinates(-.5, -.5, columnas+.5, filas+.5)
pantalla.delay(0)
tortuga = Turtle()
tortuga.hideturtle()
símbolo = crea_matriz(filas, columnas)
tablero = crea_matriz(filas, columnas)

inicializa_tablero(tablero)
rellena_símbolos(símbolo)
dibuja_tablero(tablero, símbolo)

pantalla.onclick(clic)

pantalla.mainloop()
```

Observa, además, cómo hemos cambiado la última línea del programa principal. Ya no interesa que la ejecución del programa finalice cuando se hace clic en cualquier lugar de la pantalla, pues vamos a tratar los clics con nuestro código. El método *pantalla.mainloop* hace que el programa entre en un bucle de control propio que le obliga a esperar eventos y tratarlos cuando se producen.

La mayor complicación del código va a la función *clic*. Vamos con ella:

memorion.py
```
def clic(x, y):
    global tablero, símbolo, temporal1, temporal2
    [j, i] = [int(x), int(y)]
    if 0 <= i < len(símbolo) and 0 <= j < len(símbolo[0]):
        if tablero[i][j] == CeldaCerrada:
            if temporal1 == None:
                temporal1 = [i, j]
                tablero[i][j] = CeldaTemporalmenteAbierta
            else:
                temporal2 = [i, j]
                tablero[i][j] = CeldaTemporalmenteAbierta
            dibuja_celda(tablero, símbolo, i, j)
            if temporal2 != None:
                if símbolo[temporal1[0]][temporal1[1]] \
                  == símbolo[temporal2[0]][temporal2[1]]:
                    tablero[temporal1[0]][temporal1[1]] = CeldaAbierta
                    tablero[temporal2[0]][temporal2[1]] = CeldaAbierta
                else:
                    tablero[temporal1[0]][temporal1[1]] = CeldaCerrada
                    tablero[temporal2[0]][temporal2[1]] = CeldaCerrada
                dibuja_celda(tablero, símbolo, temporal1[0], temporal1[1])
                dibuja_celda(tablero, símbolo, temporal2[0], temporal2[1])
                temporal1 = None
                temporal2 = None
        dibuja_celda(tablero, símbolo, i, j)

...
temporal1 = None
temporal2 = None
```

Nos estamos apoyando en dos variables, *temporal1* y *temporal2*, que almacenan las coordenadas de las celdas descubiertas temporalmente. Cuando *temporal1* vale None, es que no hay ninguna celda temporalmente abierta; y cuando *temporal1* tiene otro valor y *temporal2* es None, solo hay una celda temporalmente abierta. Tras comprobar que la pulsación tiene lugar en una celda válida y que esta está tapada, vemos si es la primera celda de la pareja que vamos a descubrir o si, por el contrario, es la segunda. En cualquier caso memorizamos sus coordenadas. Si es la segunda, comprobamos si las dos celdas descubiertas contienen la misma letra. Si es así, las marcamos como descubiertas. Si no, las dejamos como estaban y dejamos de considerar a ambas como temporalmente descubiertas. Finalmente, actualizamos el dibujo en pantalla. Lee el cuerpo de la función paso a paso para asegurarte de que lo entiendes.

Si ejecutas el programa verás que tiene un pequeño fallo: cuando pulsas en la segunda casilla cuya letra quieres ver, no da tiempo material de verla, pues la celda se redibuja como cerrada instantáneamente. Hemos de introducir algún retardo. Hay un modo elegante de hacer una pausa: con la función *sleep* del módulo *time*. Si llamamos a la función con un número en coma flotante, el programa se detendrá ese número de segundos al ejecutarla:

memorion.py
```
from time import sleep
...

```

```
def clic(x, y):
    global tablero, símbolo, temporal1, temporal2
    [j, i] = [int(x), int(y)]
    if 0 <= i < len(símbolo) and 0 <= j < len(símbolo[0]):
        if tablero[i][j] == CeldaCerrada:
            if temporal1 == None:
                temporal1 = [i, j]
                tablero[i][j] = CeldaTemporalmenteAbierta
            else:
                temporal2 = [i, j]
                tablero[i][j] = CeldaTemporalmenteAbierta
            dibuja_celda(tablero, símbolo, i, j)
            if temporal2 != None:
                if símbolo[temporal1[0]][temporal1[1]] \
                  == símbolo[temporal2[0]][temporal2[1]]:
                    tablero[temporal1[0]][temporal1[1]] = CeldaAbierta
                    tablero[temporal2[0]][temporal2[1]] = CeldaAbierta
                else:
                    sleep(0.5)
                    tablero[temporal1[0]][temporal1[1]] = CeldaCerrada
                    tablero[temporal2[0]][temporal2[1]] = CeldaCerrada
                dibuja_celda(tablero, símbolo, temporal1[0], temporal1[1])
                dibuja_celda(tablero, símbolo, temporal2[0], temporal2[1])
                temporal1 = None
                temporal2 = None
        dibuja_celda(tablero, símbolo, i, j)

...
temporal1 = None
temporal2 = None
```

Ya casi está. Nos falta controlar el final de la partida —todas las celdas están abiertas— y, ya puestos, pulir un detalle: cuando se está ejecutando la función *clic* hemos de desconectar momentáneamente el tratamiento de eventos. Si no lo hacemos, es posible que se ejecute una llamada a *clic* mientras hay otra llamada a *clic* en ejecución. El resultado puede ser desastroso. Este es el programa final:

```
memorion.py
from random import randrange
from turtle import Screen, Turtle
from time import sleep

CeldaCerrada = 0
CeldaAbierta = 1
CeldaTemporalmenteAbierta = 2

def crea_matriz(filas, columnas):
    matriz = []
    for i in range(filas):
        matriz.append([None] * columnas)
    return matriz

def rellena_símbolos(símbolo): # Tercera versión.
    numsímbolo = 0
    for i in range(len(símbolo)):
        for j in range(len(símbolo[0])):
            símbolo[i][j] = chr(ord('a') + numsímbolo // 2)
            numsímbolo += 1

    for i in range(1000):
        [f1, c1] = [randrange(len(símbolo)), randrange(len(símbolo[0]))]
```

```
        [f2, c2] = [randrange(len(símbolo)), randrange(len(símbolo[0]))]
        tmp = símbolo[f1][c1]
        símbolo[f1][c1] = símbolo[f2][c2]
        símbolo[f2][c2] = tmp

def inicializa_tablero(tablero):
    for i in range(len(tablero)):
        for j in range(len(tablero[0])):
            tablero[i][j] = CeldaCerrada

def dibuja_tablero(tablero, símbolo):
    for i in range(len(símbolo)):
        for j in range(len(símbolo[0])):
            dibuja_celda(tablero, símbolo, i, j)

def dibuja_celda(baldosa, símbolo, i, j):
    global tortuga
    tortuga.penup()
    tortuga.goto(j+.5, i)
    tortuga.begin_fill()
    if baldosa[i][j] == CeldaCerrada:
        tortuga.fillcolor('blue')
        tortuga.circle(.5)
    elif baldosa[i][j] == CeldaAbierta:
        tortuga.fillcolor('white')
        tortuga.circle(.5)
        tortuga.goto(j+.5, i+.25)
        tortuga.write(símbolo[i][j])
    else:
        tortuga.fillcolor('yellow')
        tortuga.circle(.5)
        tortuga.goto(j+.5, i+.25)
        tortuga.write(símbolo[i][j])
    tortuga.end_fill()
    tortuga.pendown()

def clic(x, y):
    global pantalla, tablero, símbolo, temporal1, temporal2
    pantalla.onclick(None)
    [j, i] = [int(x), int(y)]
    if 0 <= i < len(tablero) and 0 <= j < len(tablero[0]):
        if tablero[i][j] == CeldaCerrada:
            if temporal1 == None:
                temporal1 = [i, j]
                tablero[i][j] = CeldaTemporalmenteAbierta
            else:
                temporal2 = [i, j]
                tablero[i][j] = CeldaTemporalmenteAbierta
            dibuja_celda(tablero, símbolo, i, j)
            if temporal2 != None:
                if símbolo[temporal1[0]][temporal1[1]] \
                  == símbolo[temporal2[0]][temporal2[1]]:
                    tablero[temporal1[0]][temporal1[1]] = CeldaAbierta
                    tablero[temporal2[0]][temporal2[1]] = CeldaAbierta
                else:
                    sleep(0.5)
                    tablero[temporal1[0]][temporal1[1]] = CeldaCerrada
                    tablero[temporal2[0]][temporal2[1]] = CeldaCerrada
                dibuja_celda(tablero, símbolo, temporal1[0], temporal1[1])
                dibuja_celda(tablero, símbolo, temporal2[0], temporal2[1])
                temporal1 = None
```

```
                temporal2 = None
        dibuja_celda(tablero, símbolo, i, j)

    if todas_abiertas(tablero):
        pantalla.bye()
    else:
        pantalla.onclick(clic)

def todas_abiertas(tablero):
    for i in range(len(tablero)):
        for j in range(len(tablero[0])):
            if tablero[i][j] == CeldaCerrada:
                return False
    return True

# Programa principal
filas = 4
columnas = 6

pantalla = Screen()
pantalla.setup(columnas*50, filas*50)
pantalla.screensize(columnas*50, filas*50)
pantalla.setworldcoordinates(-.5, -.5, columnas+.5, filas+.5)
pantalla.delay(0)
tortuga = Turtle()
tortuga.hideturtle()
símbolo = crea_matriz(filas, columnas)
tablero = crea_matriz(filas, columnas)

temporal1 = None
temporal2 = None
inicializa_tablero(tablero)
rellena_símbolos(símbolo)
dibuja_tablero(tablero, símbolo)

pantalla.onclick(clic)

pantalla.mainloop()
```

▶ 338 Modifica Memorión para que se ofrezca al usuario jugar con tres niveles de dificultad:

- Fácil: tablero de 3×4.
- Normal: tablero de 4×6.
- Difícil: tablero de 6×8.

▶ 339 Implementa Memorión3, una variante de Memorión en la que hay que emparejar grupos de 3 letras iguales. (Asegúrate de que el número de casillas de la matriz sea múltiplo de 3).

▶ 340 Construye el programa del Buscaminas inspirándote en la forma en que hemos desarrollado el juego Memorión. Te damos unas pistas para ayudarte en la implementación:

- Crea una matriz cuyas casillas contengan el valor True o False. El primer valor indica que hay una mina en esa casilla. Ubica las minas al azar. El número de minas dependerá de la dificultad del juego.
- Crea una matriz que contenga el número de minas que rodean a cada casilla. Calcula esos valores a partir de la matriz de minas. Ojo con las casillas «especiales»: el número de vecinos de las casillas de los bordes requiere un cuidado especial.

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

- Dibuja las minas y baldosas que las tapan. Define adecuadamente el sistema de coordenadas del lienzo.
- El programa principal es un bucle similar al de Memorión. El bucle principal finaliza cuando hay una coincidencia total entre la matriz de minas y la matriz de marcas puestas por el usuario.
- Cada vez que se pulse el botón del ratón, destruye la baldosa correspondiente. Si esta escondía una mina, la partida ha acabado y el jugador ha muerto. Si no, crea un objeto gráfico (texto) que muestre el número de minas vecinas a esa casilla.

▶ 341 Modifica el Buscaminas para que cada vez que se pulse con el primer botón en una casilla con cero bombas vecinas, se marquen todas las casillas alcanzables desde esta y que no tienen bomba. (Este ejercicio es difícil. Piensa bien en la estrategia que has de seguir).

▶ 342 Diseña un programa que permita jugar a dos personas al tres en raya.

▶ 343 Diseña un programa que permita jugar al tres en raya enfrentando a una persona al ordenador. Cuando el ordenador empiece una partida, debe ganarla siempre. (Este ejercicio es difícil. Si no conoces la estrategia ganadora, búscala en Internet).

▶ 344 Diseña un programa que permita que dos personas jueguen a las damas. El programa debe verificar que todos los movimientos son válidos.

▶ 345 Diseña un programa que permita que dos personas jueguen al ajedrez. El programa debe verificar que todos los movimientos son válidos.

6.6. Ejemplos

Vamos ahora a desarrollar unos cuantos ejemplos de programas con funciones. Así pondremos en práctica lo aprendido.

6.6.1. Integración numérica

Vamos a implementar un programa de integración numérica que aproxime el valor de

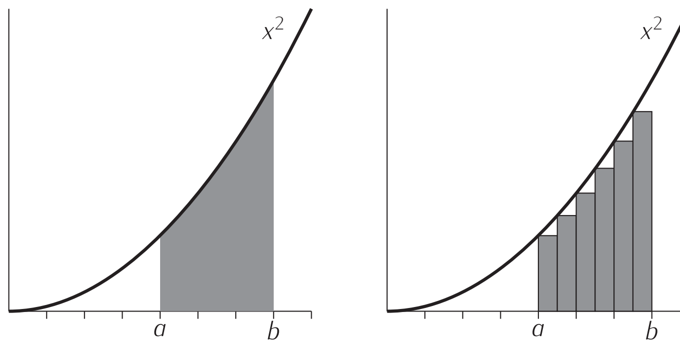
$$\int_a^b x^2\,dx$$

con la fórmula

$$\sum_{i=0}^{n-1} \Delta x \cdot (a + i \cdot \Delta x)^2,$$

donde $\Delta x = (b - a)/n$. El valor de n lo proporcionamos nosotros: a mayor valor de n, mayor precisión en la aproximación. Este método de aproximación de integrales se basa en el cálculo del área de una serie de rectángulos.

En la gráfica de la izquierda de la figura que aparece a continuación se marca en gris la región cuya área corresponde al valor de la integral de x^2 entre a y b. En la gráfica de la derecha se muestra en gris el área de cada uno de los 6 rectángulos ($n = 6$) utilizados en la aproximación. La suma de las 6 áreas es el resultado de nuestra aproximación. Si en lugar de 6 rectángulos usásemos 100, el valor calculado sería más aproximado al real.



La función Python que vamos a definir se denominará *integral_x2* y necesita tres datos de entrada: el extremo izquierdo del intervalo (a), el extremo derecho (b) y el número de rectángulos con los que se efectúa la aproximación (n).

La cabecera de la definición de la función será, pues, de la siguiente forma:

```
integral.py
def integral_x2(a, b, n):
    ...
```

¿Qué ponemos en el cuerpo? Pensemos. En el fondo, lo que se nos pide no es más que el cálculo de un sumatorio. Los elementos que participan en el sumatorio son un tanto complicados, pero esta complicación no afecta a la forma general de cálculo de un sumatorio. Los sumatorios se calculan siguiendo un patrón que ya hemos visto con anterioridad:

```
integral.py
def integral_x2(a, b, n):
    sumatorio = 0.0
    for i in range(n):
        sumatorio += lo que sea
```

Ese «*lo que sea*» es, precisamente, la fórmula que aparece en el sumatorio. En nuestro caso, ese fragmento del cuerpo de la función será así:

```
integral.py
def integral_x2(a, b, n):
    sumatorio = 0.0
    for i in range(n):
        sumatorio += deltax * (a + i * deltax) ** 2
```

Mmmm... En el bucle hacemos uso de una variable *deltax* que, suponemos, tiene el valor de Δx. Así pues, habrá que calcular previamente su valor:

```
integral.py
def integral_x2(a, b, n):
    deltax = (b-a) / n
    sumatorio = 0.0
    for i in range(n):
        sumatorio += deltax * (a + i * deltax) ** 2
```

La variable *deltax* (al igual que *i* y *sumatorio*) es una variable local.

Ya casi está. Faltará añadir una línea: la que devuelve el resultado.

```
integral.py
def integral_x2(a, b, n):
    deltax = (b-a) / n
    sumatorio = 0.0
    for i in range(n):
```

```
5          sumatorio += deltax * (a + i * deltax) ** 2
6      return sumatorio
```

¿Hecho? Repasemos, a ver si todo está bien. Fíjate en la línea 2. Esa expresión puede dar problemas si n vale 0. Debemos evitar la división por cero. Si n vale cero, el resultado de la integral será 0.

integral.py
```
1  def integral_x2(a, b, n):
2      if n == 0:
3          sumatorio = 0.0
4      else:
5          deltax = (b-a) / n
6          sumatorio = 0.0
7          for i in range(n):
8              sumatorio += deltax * (a + i * deltax) ** 2
9      return sumatorio
```

Ya podemos utilizar nuestra función:

integral.py
```
1  def integral_x2(a, b, n):
2      if n == 0:
3          sumatorio = 0.0
4      else:
5          deltax = (b-a) / n
6          sumatorio = 0.0
7          for i in range(n):
8              sumatorio += deltax * (a + i * deltax) ** 2
9      return sumatorio
10
11 inicio = float(input('Inicio del intervalo: '))
12 final = float(input('Final del intervalo: '))
13 partes = int(input('Número de rectángulos: '))
14
15 print('La integral de x**2 entre {0} y {1}'.format(inicio, final), end='')
16 print('vale aproximadamente {0}'.format(integral_x2(inicio, final, partes)))
```

En la línea 16 llamamos a *integral_x2* con los argumentos *inicio*, *final* y *partes*, variables cuyo valor nos suministra el usuario en las líneas 11–13. Recuerda que cuando llamamos a una función, Python asigna a cada parámetro el valor de un argumento siguiendo el orden de izquierda a derecha. Así, el parámetro a recibe el valor que contiene el argumento *inicio*, el parámetro b recibe el valor que contiene el argumento *final* y el parámetro n recibe el valor que contiene el argumento *partes*. No importa cómo se llama cada argumento. Una vez se han hecho esas asignaciones, empieza la ejecución de la función.

6.6.2. Aproximación de la exponencial de un número real

Vamos a desarrollar una función que calcule el valor de e^a, siendo a un número real, con una restricción: no podemos utilizar el operador de exponenciación `**`.

Si a fuese un número natural, sería fácil efectuar el cálculo:

exponencial.py
```
from math import e

def exponencial(a):
    exp = 1.0
    for i in range(a):
        exp *= e
    return exp
```

Un método de integración genérico

El método de integración que hemos implementado presenta un inconveniente: solo puede usarse para calcular la integral definida de una sola función: $f(x) = x^2$. Si queremos integrar, por ejemplo, $g(x) = x^3$, tendremos que codificar otra vez el método y cambiar una línea. ¿Y por una sóla línea hemos de volver a escribir otras ocho?

Analiza este programa:

integracion_generica.py
```
1  def cuadrado(x):
2      return x**2
3
4  def cubo(x):
5      return x**3
6
7  def integral_definida(f, a, b, n):
8      if n == 0:
9          sumatorio = 0.0
10     else:
11         deltax = (b-a) / n
12         sumatorio = 0.0
13         for i in range(n):
14             sumatorio += deltax * f(a + i * deltax)
15     return sumatorio
16
17 a = 1
18 b = 2
19 print('Integración␣entre␣{0}␣y␣{1}'.format(a, b))
20 print('Integral␣de␣x**2:', integral_definida(cuadrado, a, b, 100))
21 print('Integral␣de␣x**3:', integral_definida(cubo, a, b, 100))
```

¡Podemos pasar funciones como argumentos! En la línea 20 calculamos la integral de x^2 entre 1 y 2 (con 100 rectángulos) y en la línea 21, la de x^3. Hemos codificado una sola vez el método de integración y es, en cierto sentido, «genérico»: puede integrar cualquier función.

Pon atención a este detalle: cuando pasamos la función como parámetro, no usamos paréntesis con argumentos; solo pasamos el nombre de la función. El nombre de la función es una variable. ¿Y qué contiene? Contiene una referencia a una zona de memoria en la que se encuentran las instrucciones que hemos de ejecutar al llamar a la función. Leer ahora el cuadro «Los paréntesis son necesarios» puede ayudarte a entender esta afirmación.

▶ 346 ¿Y si a pudiera tomar valores enteros negativos? Diseña una función *exponencial* que trate también ese caso. (Recuerda que $e^{-a} = 1/e^a$).

Pero siendo a un número real (bueno, un flotante), no nos vale esa aproximación. Refrescando conocimientos matemáticos, vemos que podemos calcular el valor de e^a para a real con la siguiente fórmula:

$$e^a = 1 + a + \frac{a^2}{2} + \frac{a^3}{3!} + \frac{a^4}{4!} + \cdots + \frac{a^k}{k!} + \cdots = \sum_{n=0}^{\infty} \frac{a^n}{n!}.$$

La fórmula tiene un número infinito de sumandos, así que no la podemos codificar en Python. Haremos una cosa: diseñaremos una función que aproxime el valor de e^a con tantos sumandos como nos indique el usuario.

Vamos con una primera versión:

exponencial.py
```
def exponencial(a, n):
    sumatorio = 0.0
    for k in range(n):
```

```
        sumatorio += a**k / (k!)
    return sumatorio
```

Mmmm. Mal. Por una parte, nos han prohibido usar el operador `**`, así que tendremos que efectuar el correspondiente cálculo de otro modo. Recuerda que

$$a^k = \prod_{i=1}^{k} a.$$

exponencial.py

```
def exponencial(a, n):
    sumatorio = 0.0
    for k in range(n):
        # Cálculo de a elevado a k.
        numerador = 1.0
        for i in range(1, k+1):
            numerador *= a
        # Adición de nuevo sumando al sumatorio.
        sumatorio += numerador / k!
    return sumatorio
```

Y por otra parte, no hay operador factorial en Python. Tenemos que calcular el factorial explícitamente. Recuerda que

$$k! = \prod_{i=1}^{k} i.$$

Corregimos el programa anterior:

exponencial.py

```
def exponencial(a, n):
    sumatorio = 0.0
    for k in range(n):
        # Cálculo de a elevado a k.
        numerador = 1.0
        for i in range(1, k+1):
            numerador *= a
        # Cálculo de k factorial.
        denominador = 1.0
        for i in range(1, k+1):
            denominador *= i
        # Adición de nuevo sumando al sumatorio.
        sumatorio += numerador / denominador
    return sumatorio
```

Y ya está. La verdad es que no queda muy legible. Analiza esta otra versión:

exponencial.py

```
def elevado(a, k):
    productorio = 1.0
    for i in range(1, k+1):
        productorio *= a
    return productorio

def factorial(k):
    productorio = 1.0
    for i in range(1, k+1):
        productorio *= i
    return productorio
```

```
def exponencial(a, n):
    sumatorio = 0.0
    for k in range(n):
        sumatorio += elevado(a, k) / factorial(k)
    return sumatorio
```

Esta versión es mucho más elegante que la anterior, e igual de correcta. Al haber separado el cálculo de la exponenciación y del factorial en sendas funciones hemos conseguido que la función *exponencial* sea mucho más legible.

▶ 347 ¿Es correcta esta otra versión?

```
exponencial.py
def elevado(a, k):
    productorio = 1.0
    for i in range(k):
        productorio *= a
    return productorio

def factorial(k):
    productorio = 1.0
    for i in range(2, k+1):
        productorio *= i
    return productorio

def exponencial(a, n):
    sumatorio = 0.0
    for k in range(n):
        sumatorio += elevado(a, k) / factorial(k)
    return sumatorio
```

▶ 348 Las funciones seno y coseno se pueden calcular así

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!}$$

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!}$$

Diseña sendas funciones *seno* y *coseno* para aproximar, respectivamente, el seno y el coseno de x con n términos del sumatorio correspondiente.

El método de cálculo que hemos utilizado en la función *exponencial* es ineficiente: el término `elevado(a, k) / factorial(k)` resulta costoso de calcular. Imagina que nos piden calcular `exponencial(a, 8)`. Se producen las siguientes llamadas a *elevado* y *factorial*:

- `elevado(a, 0)` y `factorial(0)`.
- `elevado(a, 1)` y `factorial(1)`.
- `elevado(a, 2)` y `factorial(2)`.
- `elevado(a, 3)` y `factorial(3)`.
- `elevado(a, 4)` y `factorial(4)`.
- `elevado(a, 5)` y `factorial(5)`.

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

- *elevado*(*a*, 6) y *factorial*(6).
- *elevado*(*a*, 7) y *factorial*(7).

Estas llamadas esconden una repetición de cálculos que resulta perniciosa para la velocidad de ejecución del cálculo. Cada llamada a una de esas rutinas supone iterar un bucle, cuando resulta innecesario si aplicamos un poco de ingenio. Fíjate en que se cumplen estas dos relaciones:

$$\text{elevado}(a, n) = a * \text{elevado}(a, n-1), \qquad \text{factorial}(n) = n * \text{factorial}(n-1),$$

para todo n mayor que 0. Si n vale 0, tanto *elevado*(*a*, *n*) como *factorial*(*n*) valen 1. Este programa te muestra el valor de *elevado*(2, *n*) para n entre 0 y 7:

elevado_rapido.py
```python
a = 2
valor = 1
print('elevado({0}, {1}) = {2}'.format(a, 0, valor))
for n in range(1, 8):
    valor = a * valor
    print('elevado({0}, {1}) = {2}'.format(a, n, valor))
```

```
elevado(2, 0) = 1
elevado(2, 1) = 2
elevado(2, 2) = 4
elevado(2, 3) = 8
elevado(2, 4) = 16
elevado(2, 5) = 32
elevado(2, 6) = 64
elevado(2, 7) = 128
```

▶ 349 Diseña un programa similar que muestre el valor de *factorial*(*n*) para n entre 0 y 7.

Explotemos esta forma de calcular esa serie de valores en el cómputo de *exponencial*:

exponencial.py
```python
def exponencial(a, n):
    numerador = 1
    denominador = 1
    sumatorio = 1.0
    for k in range(1, n):
        numerador = a * numerador
        denominador = k * denominador
        sumatorio += numerador / denominador
    return sumatorio
```

▶ 350 Modifica las funciones que has propuesto como solución al ejercicio 348 aprovechando las siguientes relaciones, válidas para n mayor que 0:

$$\frac{(-1)^n x^{2n+1}}{(2n+1)!} = \frac{-x^2}{(2n+1)\cdot 2n} \cdot \frac{(-1)^{n-1} x^{2(n-1)+1}}{(2(n-1)+1)!},$$

$$\frac{(-1)^n x^{2n}}{(2n)!} = \frac{-x^2}{2n\cdot(2n-1)} \cdot \frac{(-1)^{n-1} x^{2(n-1)}}{(2(n-1))!}.$$

Cuando n vale 0, tenemos:

$$\frac{(-1)^0 x^1}{1!} = x, \qquad \frac{(-1)^0 x^0}{0!} = 1.$$

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

Resolvamos ahora un problema ligeramente diferente: vamos a aproximar e^a con tantos términos como sea preciso hasta que el último término considerado sea menor o igual que un valor ϵ dado. Lo desarrollaremos usando, de nuevo, las funciones *elevado* y *factorial*. (Enseguida te pediremos que mejores el programa con las últimas ideas presentadas). No resulta apropiado ahora utilizar un bucle **for-in**, pues no sabemos cuántas iteraciones habrá que dar hasta llegar a un $a^k/k!$ menor o igual que ϵ. Utilizaremos un bucle **while**:

exponencial.py
```
def elevado(a, k):
    productorio = 1.0
    for i in range(k):
        productorio *= a
    return productorio

def factorial(n):
    productorio = 1.0
    for i in range(1, n+1):
        productorio *=  i
    return productorio

def exponencial2(a, épsilon):
    sumatorio = 0.0
    k = 0
    término = elevado(a, k) / factorial(k)
    while término > épsilon:
        sumatorio += término
        k += 1
        término = elevado(a, k) / factorial(k)
    return sumatorio
```

▶ 351 Modifica la función *exponencial2* del programa anterior para que no se efectúen las ineficientes llamadas a *elevado* y *factorial*.

6.6.3. Cálculo de números combinatorios

Ahora vamos a diseñar una función que calcule de cuántas formas podemos escoger m elementos de un conjunto con n objetos. Recuerda que la fórmula es:

$$\binom{n}{m} = \frac{n!}{(n-m)!\,m!}$$

Esta función es fácil de codificar... ¡si reutilizamos la función *factorial* del apartado anterior!

combinaciones.py
```
def factorial(n):
    productorio = 1.0
    for i in range(1, n+1):
        productorio *= i
    return productorio

def combinaciones(n, m):
    return factorial(n) / (factorial(n-m) * factorial(m))
```

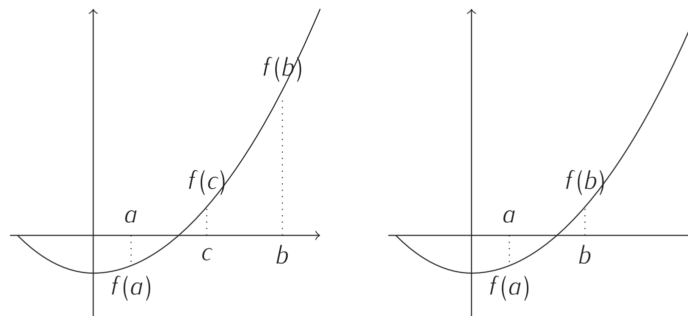
Observa cuán apropiado ha resultado que *factorial* fuera una función definida independientemente: hemos podido utilizarla en tres sitios diferentes con solo invocarla. Además, una vez diseñada la función *factorial*, podemos reutilizarla en otros programas con solo «copiar y pegar». Más adelante te enseñaremos cómo hacerlo aún más cómodamente.

6.6.4. El método de la bisección

El método de la bisección permite encontrar un cero de una función matemática $f(x)$ en un intervalo $[a, b]$ si $f(x)$ es continua en dicho intervalo y $f(a)$ y $f(b)$ son de distinto signo.

El método de la bisección consiste en dividir el intervalo en dos partes iguales. Llamemos c al punto medio del intervalo. Si el signo de $f(c)$ tiene el mismo signo que $f(a)$, aplicamos el mismo método al intervalo $[c, b]$. Si $f(c)$ tiene el mismo signo que $f(b)$, aplicamos el método de la bisección al intervalo $[a, c]$. El método finaliza cuando hallamos un punto c tal que $f(c) = 0$ o cuando la longitud del intervalo de búsqueda es menor que un ϵ determinado.

En la figura de la izquierda te mostramos el instante inicial de la búsqueda: nos piden hallar un cero de una función continua f entre a y b y ha de haberlo porque el signo de $f(a)$ es distinto del de $f(b)$. Calculamos entonces el punto medio c entre a y b. $f(c)$ y $f(b)$ presentan el mismo signo, así que el cero no se encuentra entre c y b, sino entre a y c. La figura de la derecha te muestra la nueva zona de interés: b ha cambiado su valor y ha tomado el que tenía c.



Deseamos diseñar un programa que aplique el método de la bisección a la búsqueda de un cero de la función $f(x) = x^2 - 2x - 2$ en el intervalo $[0.5, 3.5]$. No debemos considerar intervalos de búsqueda mayores que 10^{-5}.

Parece claro que implementaremos dos funciones: una para la función matemática $f(x)$ y otra para el método de la bisección. Esta última tendrá tres parámetros: los dos extremos del intervalo y el valor de ϵ que determina el tamaño del (sub)intervalo de búsqueda más pequeño que queremos considerar:

```
biseccion.py
def f(x):
    return x**2 - 2*x - 2

def bisección(a, b, épsilon):
    ...
```

El método de la bisección es un método iterativo: aplica un mismo procedimiento repetidas veces hasta satisfacer cierta condición. Utilizaremos un bucle, pero ¿un **while** o un **for-in**? Obviamente, un bucle **while**: no sabemos a priori cuántas veces iteraremos. ¿Cómo decidimos cuándo hay que volver a iterar? Hay que volver a iterar mientras no hayamos hallado el cero *y, además,* el intervalo de búsqueda sea mayor que ϵ:

```
biseccion.py
def f(x):
    return x**2 - 2*x - 2

def bisección(a, b, épsilon):
    while f(c) != 0 and b - a > épsilon:
        ...
```

Para que la primera comparación funcione c ha de tener asignado algún valor:

```
biseccion.py
def f(x):
```

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1
Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

```
    return x**2 - 2*x - 2

def bisección(a, b, épsilon):
    c = (a + b) / 2
    while f(c) != 0 and b - a > épsilon:
        ...
```

Parámetros con valor por defecto

La función *bisección* trabaja con tres parámetros. El tercero está relacionado con el margen de error que aceptamos en la respuesta. Supón que el noventa por cien de las veces trabajamos con un valor de ϵ fijo, pongamos que igual a 10^{-5}. Puede resultar pesado proporcionar explícitamente ese valor en todas y cada una de las llamadas a la función. Python nos permite proporcionar parámetros con un valor por defecto. Si damos un valor por defecto al parámetro *épsilon*, podremos llamar a la función *bisección* con tres argumentos, como siempre, o con solo dos.

El valor por defecto de un parámetro se declara en la definición de la función:

```
def bisección(a, b, épsilon=1e-5):
    ...
```

Si llamamos a la función mediante *bisección*(1, 2), es como si la hubiésemos llamado así: *bisección*(1, 2, 1e-5). Al no indicar valor para *épsilon*, Python toma su valor por defecto.

Dentro del bucle hemos de actualizar el intervalo de búsqueda:

```
biseccion.py
def f(x):
    return x**2 - 2*x - 2

def bisección(a, b, épsilon):
    c = (a + b) / 2
    while f(c) != 0 and b - a > épsilon:
        if (f(a) < 0 and f(c) < 0 ) or (f(a) > 0 and f(c) > 0):
            a = c
        elif (f(b) < 0 and f(c) < 0 ) or (f(b) > 0 and f(c) > 0):
            b = c
      ...
```

Las condiciones del **if-elif** son complicadas. Podemos simplificarlas con una idea feliz: dos números x e y tienen el mismo signo si su producto es positivo.

```
biseccion.py
def f(x):
    return x**2 - 2*x - 2

def bisección(a, b, épsilon):
    c = (a + b) / 2
    while f(c) != 0 and b - a > épsilon:
        if f(a) * f(c) > 0:
            a = c
        elif f(b) * f(c) > 0:
            b = c
        ...
```

Aún nos queda «preparar» la siguiente iteración. Si no actualizamos el valor de c, la función quedará atrapada en un bucle sin fin. ¡Ah! Y al finalizar el bucle hemos de devolver el cero de la función:

```
biseccion.py
def f(x):
    return x**2 - 2*x - 2

def bisección(a, b, épsilon):
    c = (a + b) / 2
    while f(c) != 0 and b - a > épsilon:
        if f(a) * f(c) > 0:
            a = c
        elif f(b) * f(c) > 0:
            b = c
        c = (a + b) / 2
    return c
```

Ya podemos completar el programa introduciendo el intervalo de búsqueda y el valor de ϵ:

```
biseccion.py
def f(x):
    return x**2 - 2*x - 2

def bisección(a, b, épsilon):
    c = (a + b) / 2
    while f(c) != 0 and b - a > épsilon:
        if f(a) * f(c) > 0:
            a = c
        elif f(b) * f(c) > 0:
            b = c
        c = (a + b) / 2
    return c

print('El␣cero␣está␣en:', bisección(0.5, 3.5, 1e-5))
```

▶ 352 La función *bisección* aún no está acabada del todo. ¿Qué ocurre si el usuario introduce un intervalo $[a, b]$ tal que $f(a)$ y $f(b)$ tienen el mismo signo? ¿Y si $f(a)$ o $f(b)$ valen 0? Modifica la función para que solo inicie la búsqueda cuando procede y, en caso contrario, devuelva el valor especial None. Si $f(a)$ o $f(b)$ valen cero, *bisección* devolverá el valor de a o b, según proceda.

▶ 353 Modifica el programa para que solicite al usuario los valores a, b y ϵ. El programa solo aceptará valores de a y b tales que $a < b$.

6.7. Diseño de programas con funciones

Hemos aprendido a diseñar funciones, cierto, pero puede que no tengas claro qué ventajas nos reporta trabajar con ellas. El programa de integración numérica que hemos desarrollado en la sección anterior podría haberse escrito directamente así:

```
integral.py
a = float(input('Inicio␣del␣intervalo:␣'))
b = float(input('Final␣del␣intervalo:␣'))
n = int(input('Número␣de␣rectángulos:␣'))

if n == 0:
    sumatorio = 0.0
else:
    deltax = (b - a) / n
    sumatorio = 0.0
    for i in range(n):
        sumatorio += deltax * (a + i * deltax) ** 2
```

Evita las llamadas repetidas

En nuestra última versión del programa `biseccion.py` hay una fuente de ineficiencia: `f(c)`, para un c fijo, se calcula 3 veces por iteración.

```
biseccion.py
def f(x):
    return x**2 - 2*x - 2

def bisección(a, b, épsilon):
    c = (a + b) / 2
    while f(c) != 0 and b - a > épsilon:
        if f(a) * f(c) > 0:
            a = c
        elif f(b) * f(c) > 0:
            b = c
        c = (a + b) / 2
    return c
```

Llamar a una función es costoso: se debe dedicar un tiempo a gestionar la pila de llamadas apilando una nueva trama de activación (y ocupar, en consecuencia, algo de memoria), copiar referencias a los valores de los argumentos en los parámetros formales, efectuar un cálculo que posiblemente ya hayamos hecho y devolver el valor resultante. Una optimización del programa consiste en no llamar a `f(c)` más que una vez y almacenar el resultado en una variable temporal que usaremos cada vez que deberíamos haber llamado a `f(c)`:

```
biseccion.py
def bisección(a, b, épsilon):
    c = (a + b) / 2
    fc = f(c)
    while fc != 0 and b - a > épsilon:
        if f(a) * fc > 0:
            a = c
        elif f(b) * fc > 0:
            b = c
        c = (a + b) / 2
        fc = f(c)
   return c
```

```

print('La␣integral␣de␣x**2␣entre␣',end='')
print('{0}␣y␣{1}␣es␣(aprox)␣{2}'.format(a, b, sumatorio))
```

Este programa ocupa menos líneas y hace lo mismo, ¿no? Sí, así es. Con programas pequeños como este apenas podemos apreciar las ventajas de trabajar con funciones. Imagina que el programa fuese mucho más largo y que hiciese falta aproximar el valor de la integral definida de x^2 en tres o cuatro lugares diferentes; entonces sí que sería una gran ventaja haber definido una función: habiendo escrito el procedimiento de cálculo una vez podríamos ejecutarlo cuantas veces quisiéramos mediante simples invocaciones. No solo eso, habríamos ganado en legibilidad.

6.7.1. Ahorro de tecleo

Por ejemplo, supón que en un programa deseamos leer tres números enteros y asegurarnos de que sean positivos. Podemos proceder repitiendo el bucle correspondiente tres veces:

```
lee_positivos.py
a = int(input('Dame␣un␣número␣positivo:␣'))
while a < 0:
    print('Has␣cometido␣un␣error:␣el␣número␣debe␣ser␣positivo')
    a = int(input('Dame␣un␣número␣positivo:␣'))
```

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

```

b = int(input('Dame otro número positivo: '))
while b < 0:
    print('Has cometido un error: el número debe ser positivo')
    b = int(input('Dame otro número positivo: '))

c = int(input('Dame otro número positivo: '))
while c < 0:
    print('Has cometido un error: el número debe ser positivo')
    c = int(input('Dame otro número positivo: '))
```

O podemos llamar tres veces a una función que lea un número y se asegure de que sea positivo:

lee_positivos.py

```
def lee_entero_positivo(texto):
    número = int(input(texto))
    while número < 0:
        print('Has cometido un error: el número debe ser positivo')
        número = int(input(texto))
    return número

a = lee_entero_positivo('Dame un número positivo: ')
b = lee_entero_positivo('Dame otro número positivo: ')
c = lee_entero_positivo('Dame otro número positivo: ')
```

Hemos reducido el número de líneas, así que hemos tecleado menos. Ahorrar tecleo tiene un efecto secundario beneficioso: reduce la posibilidad de cometer errores. Si hubiésemos escrito mal el procedimiento de lectura del valor entero positivo, bastaría con corregir la función correspondiente. Si en lugar de definir esa función hubiésemos replicado el código, nos tocaría corregir el mismo error en varios puntos del programa. Es fácil que, por descuido, olvidásemos corregir el error en uno de esos lugares y, sin embargo, pensásemos que el problema está solucionado.

6.7.2. Mejora de la legibilidad

No solo nos ahorramos teclear: un programa que utiliza funciones es, por regla general, más legible que uno que inserta los procedimientos de cálculo directamente donde se utilizan; bueno, eso siempre que escojas nombres de función que describan bien qué hacen estas. Fíjate en que el último programa es más fácil de leer que el anterior, pues estas tres líneas son autoexplicativas:

```
a = lee_entero_positivo('Dame un número positivo: ')
b = lee_entero_positivo('Dame otro número positivo: ')
c = lee_entero_positivo('Dame otro número positivo: ')
```

6.7.3. Algunos consejos para decidir qué debería definirse como función: análisis descendente y ascendente

Las funciones son un elemento fundamental de los programas. Ahora ya sabes *cómo* construir funciones, pero quizá no sepas *cuándo* conviene construirlas. Lo cierto es que no podemos decírtelo: no es una ciencia exacta, sino una habilidad que irás adquiriendo con la práctica. De todos modos, sí podemos darte algunos consejos.

1) Por una parte, *todos los fragmentos de programa que vayas a utilizar en más de una ocasión son buenos candidatos a definirse como funciones*, pues de ese modo evitarás tener que copiarlos en varios lugares. Evitar esas copias no solo resulta más cómodo: también reduce considerablemente la probabilidad de que cometas errores, pues acabas escribiendo menos

texto. Además, si cometes errores y has de corregirlos o si has de modificar el programa para ampliar su funcionalidad, siempre será mejor que el mismo texto no aparezca en varios lugares, sino una sola vez en una función.

2) *Si un fragmento de programa lleva a cabo una acción que puedes nombrar o describir con una sola frase, probablemente convenga convertirlo en una función.* No olvides que los programas, además de funcionar correctamente, deben ser legibles. Lo ideal es que el programa conste de una serie de definiciones de función y un programa principal breve que las use y resulte muy legible.

3) *No conviene que las funciones que definas sean muy largas.* En general, una función debería ocupar menos de 30 o 40 líneas (aunque siempre hay excepciones). *Una función no solo debería ser breve, además debería hacer una única cosa... y hacerla bien.* Deberías ser capaz de describir con *una sola frase* lo que hace cada una de tus funciones. Si una función hace tantas cosas que explicarlas todas cuesta mucho, probablemente harías bien en dividir tu función en funciones más pequeñas y simples. Recuerda que puedes llamar a una función desde otra.

El proceso de identificar acciones complejas y dividirlas en acciones más sencillas se conoce como *estrategia de diseño descendente* (en inglés, «top-down»). La forma de proceder es esta:

- analiza primero qué debe hacer tu programa y haz un esquema que explicite las diferentes acciones que debe efectuar, pero sin entrar en el detalle de cómo debe efectuarse cada una de ellas;
- define una posible función por cada una de esas acciones;
- analiza entonces cada una de esas acciones y mira si aún son demasiado complejas; si es así, aplica el mismo método hasta que obtengas funciones más pequeñas y simples.

Una estrategia de diseño alternativa recibe el calificativo de *ascendente* (en inglés, «bottom-up») y consiste en lo contrario:

- detecta algunas de las acciones más simples que necesitarás en tu programa y escribe pequeñas funciones que las implementen;
- combina estas acciones en otras más complejas y crea nuevas funciones para ellas;
- sigue hasta llegar a una o unas pocas funciones que resuelven el problema.

Ahora que empiezas a programar resulta difícil que seas capaz de anticiparte y detectes a simple vista qué pequeñas funciones te irán haciendo falta y cómo combinarlas apropiadamente. Será más efectivo que empieces siguiendo la metodología descendente: ve dividiendo cada problema en subproblemas más y más sencillos que, al final, se combinarán para dar solución al problema original. Cuando tengas mucha más experiencia, probablemente descubrirás que al programar sigues una estrategia híbrida, ascendente y descendente a la vez. Todo llega. Paciencia.

6.8. Recursión

Desde una función puedes llamar a otras funciones. Ya lo hemos hecho en los ejemplos que hemos estudiado, pero ¿qué ocurriría si una función llamara a otra y esta, a su vez, llamara a la primera? O de modo más inmediato, ¿qué pasaría si una función se llamara a sí misma?

Una función que se llama a sí misma, directa o indirectamente, es una *función recursiva*. La recursión es un potente concepto con el que se pueden expresar ciertos procedimientos de cálculo muy elegantemente. No obstante, al principio cuesta un poco entender las funciones recursivas... y un poco más diseñar nuestras propias funciones recursivas. La recursión es un concepto difícil cuando estás aprendiendo a programar. No te asustes si este material se te resiste más que el resto.

6.8.1. Cálculo recursivo del factorial

Empezaremos por presentar y estudiar una función recursiva: el cálculo recursivo del factorial de un número natural. Partiremos de la siguiente definición matemática, válida para valores positivos de n:

$$n! = \begin{cases} 1, & \text{si } n = 0 \text{ o } n = 1; \\ n \cdot (n-1)!, & \text{si } n > 1. \end{cases}$$

Es una definición de factorial un tanto curiosa: ¡se define en términos de sí misma! El segundo de sus dos casos dice que para conocer el factorial de n hay que conocer el factorial de $n-1$ y multiplicarlo por n. Entonces, ¿cómo calculamos el factorial de $n-1$? En principio, conociendo antes el valor del factorial de $n-2$ y multiplicando ese valor por $n-1$. ¿Y el de $n-2$? Pues del mismo modo… y así hasta que acabemos por preguntarnos cuánto vale el factorial de 1. En ese momento no necesitaremos hacer más cálculos: el primer caso de la fórmula nos dice que 1! vale 1.

Vamos a plasmar esta idea en una función Python:

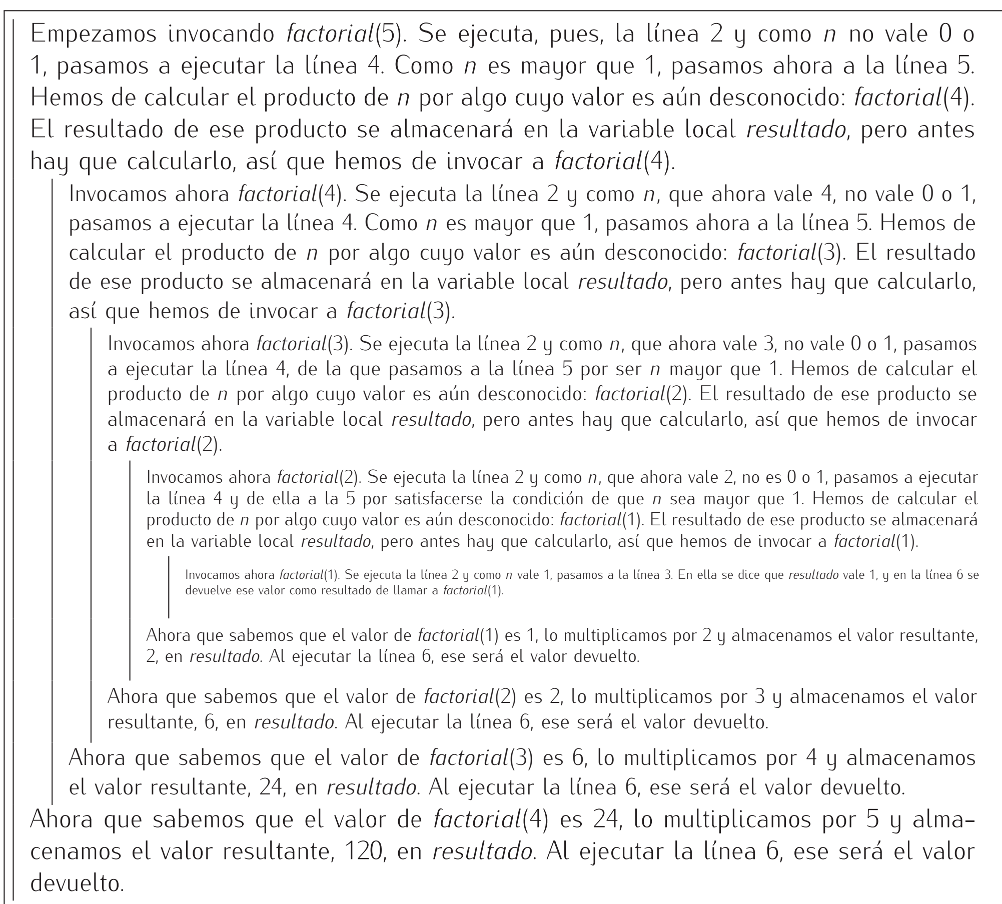
factorial.py
```
def factorial(n):
    if n == 0 or n == 1:
        resultado = 1
    elif n > 1:
        resultado = n * factorial(n-1)
    return resultado
```

Compara la fórmula matemática y la función Python. No son tan diferentes. Python nos fuerza a decir lo mismo de otro modo, es decir, con otra *sintaxis*. Más allá de las diferencias de forma, ambas definiciones son idénticas.

Para entender la recursión, nada mejor que verla en funcionamiento. La figura 6.1 te muestra paso a paso qué ocurre si solicitamos el cálculo del factorial de 5. Estudia bien la figura. Con el anidamiento de cada uno de los pasos pretendemos ilustrar que el cálculo de cada uno de los factoriales tiene lugar mientras el anterior aún está pendiente de completarse. En el nivel más interno, *factorial*(5) está pendiente de que acabe *factorial*(4), que a su vez está pendiente de que acabe *factorial*(3), que a su vez está pendiente de que acabe *factorial*(2), que a su vez está pendiente de que acabe *factorial*(1). Cuando *factorial*(1) acaba, pasa el valor 1 a *factorial*(2), que a su vez pasa el valor 2 a *factorial*(3), que a su vez pasa el valor 6 a *factorial*(4), que a su vez pasa el valor 24 a *factorial*(5), que a su vez devuelve el valor 120.

De acuerdo, la figura 6.1 describe con mucho detalle lo que ocurre, pero es difícil de seguir y entender. Veamos si la figura 6.2 te es de más ayuda. En esa figura también se describe paso a paso lo que ocurre al calcular el factorial de 5, solo que con la ayuda de unos muñecos.

- En el paso 1, le encargamos a Amadeo que calcule el factorial de 5. Él no sabe calcular el factorial de 5, a menos que alguien le diga lo que vale el factorial de 4.
- En el paso 2, Amadeo llama a un hermano clónico suyo, Benito, y le pide que calcule el factorial de 4. Mientras Benito intenta resolver el problema, Amadeo se echa a dormir (paso 3).
- Benito tampoco sabe resolver directamente factoriales tan complicados, así que llama a su clon Ceferino en el paso 4 y le pide que calcule el valor del factorial de 3. Mientras, Benito se echa a dormir (paso 5).
- La cosa sigue igual un ratillo: Ceferino llama al clon David y David a Eduardo. Así llegamos al paso 9 en el que Amadeo, Benito, Ceferino y David están durmiendo y Eduardo se pregunta cuánto valdrá el factorial de 1.
- En el paso 10 vemos que Eduardo cae en la cuenta de que el factorial de 1 es muy fácil de calcular: vale 1.



Empezamos invocando *factorial*(5). Se ejecuta, pues, la línea 2 y como *n* no vale 0 o 1, pasamos a ejecutar la línea 4. Como *n* es mayor que 1, pasamos ahora a la línea 5. Hemos de calcular el producto de *n* por algo cuyo valor es aún desconocido: *factorial*(4). El resultado de ese producto se almacenará en la variable local *resultado*, pero antes hay que calcularlo, así que hemos de invocar a *factorial*(4).

> Invocamos ahora *factorial*(4). Se ejecuta la línea 2 y como *n*, que ahora vale 4, no vale 0 o 1, pasamos a ejecutar la línea 4. Como *n* es mayor que 1, pasamos ahora a la línea 5. Hemos de calcular el producto de *n* por algo cuyo valor es aún desconocido: *factorial*(3). El resultado de ese producto se almacenará en la variable local *resultado*, pero antes hay que calcularlo, así que hemos de invocar a *factorial*(3).
>
>> Invocamos ahora *factorial*(3). Se ejecuta la línea 2 y como *n*, que ahora vale 3, no vale 0 o 1, pasamos a ejecutar la línea 4, de la que pasamos a la línea 5 por ser *n* mayor que 1. Hemos de calcular el producto de *n* por algo cuyo valor es aún desconocido: *factorial*(2). El resultado de ese producto se almacenará en la variable local *resultado*, pero antes hay que calcularlo, así que hemos de invocar a *factorial*(2).
>>
>>> Invocamos ahora *factorial*(2). Se ejecuta la línea 2 y como *n*, que ahora vale 2, no es 0 o 1, pasamos a ejecutar la línea 4 y de ella a la 5 por satisfacerse la condición de que *n* sea mayor que 1. Hemos de calcular el producto de *n* por algo cuyo valor es aún desconocido: *factorial*(1). El resultado de ese producto se almacenará en la variable local *resultado*, pero antes hay que calcularlo, así que hemos de invocar a *factorial*(1).
>>>
>>>> Invocamos ahora *factorial*(1). Se ejecuta la línea 2 y como *n* vale 1, pasamos a la línea 3. En ella se dice que *resultado* vale 1, y en la línea 6 se devuelve ese valor como resultado de llamar a *factorial*(1).
>>>
>>> Ahora que sabemos que el valor de *factorial*(1) es 1, lo multiplicamos por 2 y almacenamos el valor resultante, 2, en *resultado*. Al ejecutar la línea 6, ese será el valor devuelto.
>>
>> Ahora que sabemos que el valor de *factorial*(2) es 2, lo multiplicamos por 3 y almacenamos el valor resultante, 6, en *resultado*. Al ejecutar la línea 6, ese será el valor devuelto.
>
> Ahora que sabemos que el valor de *factorial*(3) es 6, lo multiplicamos por 4 y almacenamos el valor resultante, 24, en *resultado*. Al ejecutar la línea 6, ese será el valor devuelto.

Ahora que sabemos que el valor de *factorial*(4) es 24, lo multiplicamos por 5 y almacenamos el valor resultante, 120, en *resultado*. Al ejecutar la línea 6, ese será el valor devuelto.

Figura 6.1: Traza del cálculo recursivo de *factorial*(5).

- En el paso 11 Eduardo despierta a David y le comunica lo que ha averiguado: el factorial de 1! vale 1.

- En el paso 12 Eduardo nos ha abandonado: él ya cumplió con su deber. Ahora es David el que resuelve el problema que le habían encargado: 2! se puede calcular multiplicando 2 por lo que valga 1!, y Eduardo le dijo que 1! vale 1.

- En el paso 13 David despierta a Ceferino para comunicarle que 2! vale 2. En el paso 14 Ceferino averigua que 3! vale 6, pues resulta de multiplicar 3 por el valor que David le ha comunicado.

- Y así sucesivamente hasta llegar al paso 17, momento en el que Benito despierta a Amadeo y le dice que 4! vale 24.

- En el paso 18 solo queda Amadeo y descubre que 5! vale 120, pues es el resultado de multiplicar por 5 el valor de 4!, que según Benito es 24.

Una forma compacta de representar la secuencia de llamadas es mediante el denominado *árbol de llamadas*:

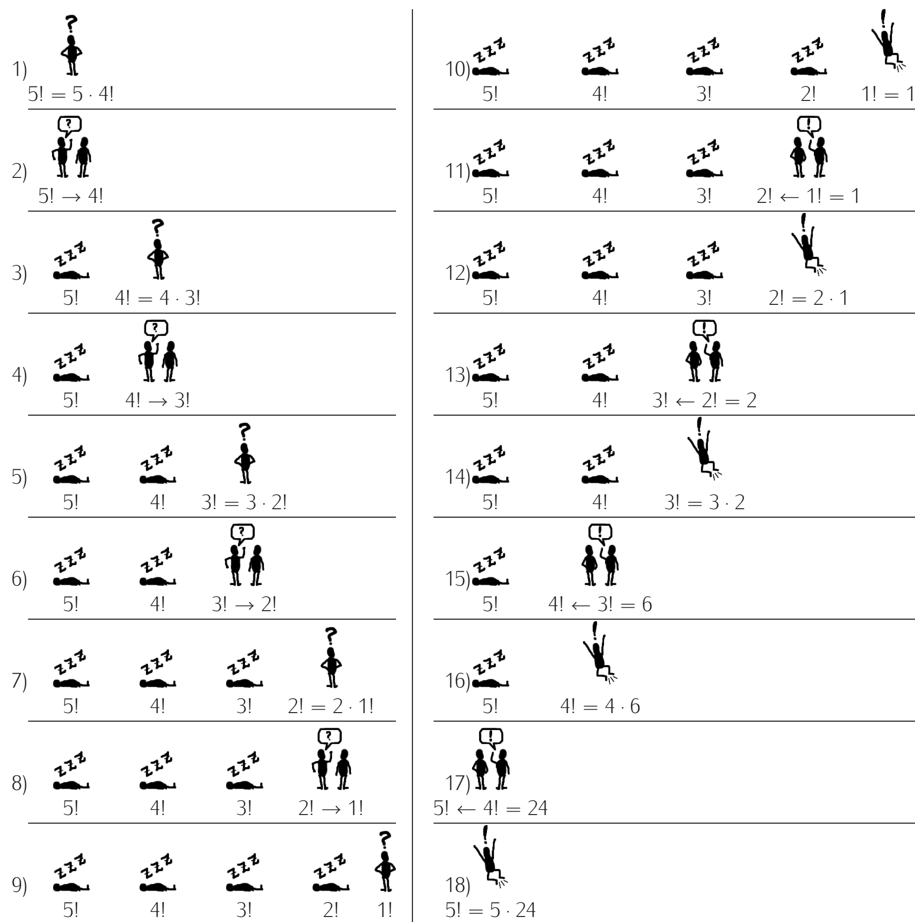
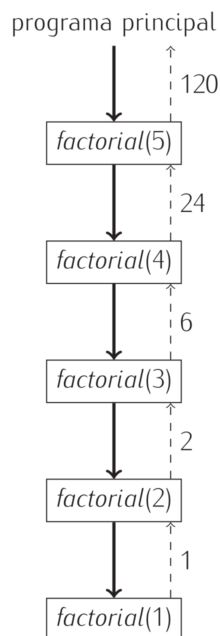


Figura 6.2: Cómic explicativo del cálculo recursivo del factorial de 5.



Los nodos del árbol de llamadas se visitan de arriba a abajo (flechas de trazo continuo) y cuando se ha alcanzado el último nodo, de abajo a arriba. Sobre las flechas de trazo discontinuo hemos representado el valor devuelto por cada llamada.

¿Recurrir o iterar?

Hemos propuesto una solución recursiva para el cálculo del factorial, pero en anteriores apartados hemos hecho ese mismo cálculo con un método iterativo. Esta función calcula el factorial iterativamente (con un bucle for-in):

factorial.py
```
def factorial(n):
    f = 1
    for i in range(1,n+1):
        f *= i
    return f
```

Pues bien, para toda función recursiva podemos encontrar otra que haga el mismo cálculo de modo iterativo. Ocurre que no siempre es fácil hacer esa conversión o que, en ocasiones, la versión recursiva es más elegante y legible que la iterativa (o, cuando menos, se parece más a la definición matemática). Por otra parte, las versiones iterativas suelen ser más eficientes que las recursivas, pues cada llamada a una función supone pagar una pequeña penalización en tiempo de cálculo y espacio de memoria, ya que se consume memoria y algo de tiempo en gestionar la pila de llamadas a función.

▶ 354 Haz una traza de la pila de llamadas a función paso a paso para *factorial*(5).

▶ 355 También podemos formular recursivamente la suma de los n primeros números naturales:

$$\sum_{i=1}^{n} i = \begin{cases} 1, & \text{si } n = 1; \\ n + \sum_{i=1}^{n-1} i, & \text{si } n > 1. \end{cases}$$

Diseña una función Python recursiva que calcule el sumatorio de los n primeros números naturales.

▶ 356 Inspirándote en el ejercicio anterior, diseña una función recursiva que, dados m y n, calcule

$$\sum_{i=m}^{n} i.$$

▶ 357 La siguiente función implementa recursivamente una comparación entre dos números naturales. ¿Qué comparación?

compara.py
```
def comparación(a, b):
    if b == 0:
        return False
    elif a == 0:
        return True
    else:
        return comparación(a-1, b-1)
```

6.8.2. Cálculo recursivo del número de bits necesarios para representar un número

Vamos con otro ejemplo de recursión. Vamos a hacer un programa que determine el número de bits necesarios para representar un número entero dado. Para pensar en términos recursivos hemos de actuar en dos pasos:

1) Encontrar uno o más casos sencillos, tan sencillos que sus respectivas soluciones sean obvias. A esos casos los llamaremos *casos base*.

Regresión infinita

Observa que una elección inapropiada de los casos base puede conducir a una recursión que no se detiene jamás. Es lo que se conoce por *regresión infinita* y es análoga a los bucles infinitos.

Por ejemplo, imagina que deseamos implementar el cálculo recursivo del factorial y diseñamos esta función errónea:

factorial.py
```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

¿Qué ocurre si calculamos con ella el factorial de 0, que es 1? Se dispara una cadena infinita de llamadas recursivas, pues el factorial de 0 llama a factorial de −1, que a su vez llama a factorial de −2, y así sucesivamente. Jamás llegaremos al caso base.

De todos modos, el computador no se quedará colgado indefinidamente: el programa acabará por provocar una excepción. ¿Por qué? Porque la pila de llamadas irá creciendo hasta ocupar toda la memoria disponible, y entonces Python indicará que se produjo un «desbordamiento de pila» (en inglés, «stack overflow»).

2) Plantear el caso general en términos de un problema similar, *pero más sencillo*. Si, por ejemplo, la entrada del problema es un número, conviene que propongas una solución en términos de un problema equivalente sobre un número más pequeño.

En nuestro problema los casos base serían 0 y 1: los números 0 y 1 necesitan un solo bit para ser representados, sin que sea necesario hacer ningún cálculo para averiguarlo. El caso general, digamos n, puede plantearse del siguiente modo: el número n puede representarse con 1 bit más que el número $n//2$ (donde la división es entera). El cálculo del número de bits necesarios para representar $n//2$ parece más sencillo que el del número de bits necesarios para representar n, pues $n//2$ es más pequeño que n.

Comprobemos que nuestro razonamiento es cierto. ¿Cuántos bits hacen falta para representar el número 5? Uno más que los necesarios para representar el 2 (que es el resultado de dividir 5 entre 2 y quedarnos con la parte entera). ¿Y para representar el número 2? Uno más que los necesarios para representar el 1. ¿Y para representar el número 1?: fácil, ese es un caso base cuya solución es 1 bit. Volviendo hacia atrás queda claro que necesitamos 2 bits para representar el número 2 y 3 bits para representar el número 5.

Ya estamos en condiciones de escribir la función recursiva:

bits.py
```
def bits(n):
    if n == 0 or n == 1:
        resultado = 1
    else:
        resultado = 1 + bits(n // 2)
    return resultado
```

▶ 358 Dibuja un árbol de llamadas que muestre paso a paso lo que ocurre cuando calculas *bits*(63).

▶ 359 Diseña una función recursiva que calcule el número de dígitos que tiene un número entero (en base 10).

6.8.3. Los números de Fibonacci

El ejemplo que vamos a estudiar ahora es el del cálculo recursivo de números de Fibonacci. Los números de Fibonacci son una secuencia de números muy particular:

F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	...
1	1	2	3	5	8	13	21	34	55	89	...

Los dos primeros números de la secuencia valen 1 y cada número a partir del tercero se obtiene sumando los dos anteriores. Podemos expresar esta definición matemáticamente así:

$$F_n = \begin{cases} 1, & \text{si } n = 1 \text{ o } n = 2; \\ F_{n-1} + F_{n-2}, & \text{si } n > 2. \end{cases}$$

Los números de Fibonacci en el mundo real

Los números de Fibonacci son bastante curiosos, pues aparecen espontáneamente en la naturaleza. Te presentamos algunos ejemplos:

- Las abejas comunes viven en colonias. En cada colonia hay una sola reina (hembra), muchas trabajadoras (hembras estériles) y algunos zánganos (machos). Los machos nacen de huevos no fertilizados, por lo que tienen madre, pero no padre. Las hembras nacen de huevos fertilizados y, por tanto, tienen padre y madre. Estudiemos el árbol genealógico de 1 zángano: tiene 1 madre, 2 abuelos (su madre tiene padre y madre), 3 bisabuelos, 5 tatarabuelos, 8 tatara-tatarabuelos, 13 tatara-tatara-tatarabuelos... Fíjate en la secuencia: 1, 1, 2, 3, 5, 8, 13, ... A partir del tercero, cada número se obtiene sumando los dos anteriores. Esta secuencia es la serie de Fibonacci.
- Muchas plantas tienen un número de pétalos que coincide con esa secuencia de números: la flor del iris tiene 3 pétalos, la de la rosa silvestre, 5 pétalos, la del dephinium, 8, la de la cineraria, 13, la de la chicoria, 21, ... Y así sucesivamente (las hay con 34, 55 y 89 pétalos).
- El número de espirales cercanas al centro de un girasol que van hacia a la izquierda y las que van hacia la derecha son, ambos, números de la secuencia de Fibonacci.
- También el número de espirales que en ambos sentidos presenta la piel de las piñas coincide con sendos números de Fibonacci.

Podríamos dar aún más ejemplos. Los números de Fibonacci aparecen por doquier. Y además, son tan interesantes desde un punto de vista matemático que hay una asociación dedicada a su estudio que edita trimestralmente una revista especializada con el título *The Fibonacci Quarterly*.

La transcripción de esta definición a una función Python es fácil:

fibonacci.py
```python
def fibonacci(n):
    if n == 1 or n == 2:
        resultado = 1
    elif n > 2:
        resultado = fibonacci(n-1) + fibonacci(n-2)
    return resultado
```

Ahora bien, entender cómo funciona *fibonacci* en la práctica puede resultar un tanto más díficil, pues el cálculo de *un* número de Fibonacci necesita conocer el resultado de *dos* cálculos adicionales (salvo en los casos base, claro está). Veámoslo con un pequeño ejemplo: el cálculo de *fibonacci*(4).

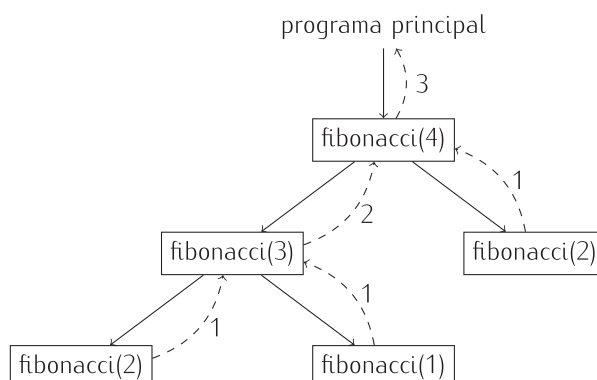
- Llamamos a *fibonacci*(4). Como n no vale ni 1 ni 2, hemos de llamar a *fibonacci*(3) y a *fibonacci*(2) para, una vez devueltos sus respectivos valores, sumarlos. Pero no se ejecutan ambas llamadas simultáneamente. Primero se llama a uno (a *fibonacci*(3)) y luego al otro (a *fibonacci*(2)).
    - Llamamos primero a *fibonacci*(3). Como n no vale ni 1 ni 2, hemos de llamar a *fibonacci*(2) y a *fibonacci*(1) para, una vez recibidos los valores que devuelven, sumarlos. Primero se llama a *fibonacci*(2), y luego a *fibonacci*(1).
        - Llamamos primero a *fibonacci*(2). Este es fácil: devuelve el valor 1.
        - Llamamos a continuación a *fibonacci*(1). Este también es fácil: devuelve el valor 1.

Ahora que sabemos que *fibonacci*(2) devuelve un 1 y que *fibonacci*(1) devuelve un 1, sumamos ambos valores y devolvemos un 2. (Recuerda que estamos ejecutando una llamada a *fibonacci*(3)).

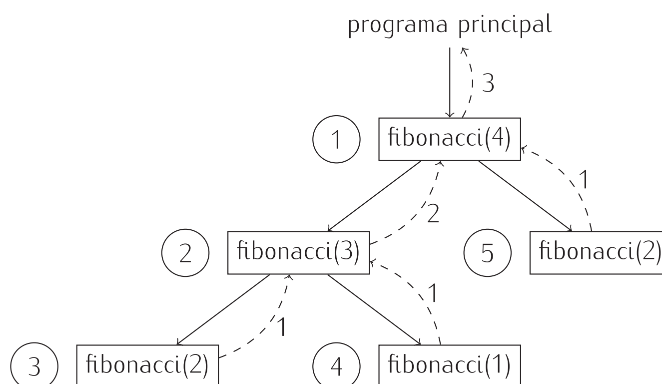
- Y ahora llamamos a *fibonacci*(2), que inmediatamente devuelve un 1.

Ahora que sabemos que *fibonacci*(3) devuelve un 2 y que *fibonacci*(2) devuelve un 1, sumamos ambos valores y devolvemos un 3. (Recuerda que estamos ejecutando una llamada a *fibonacci*(4)).

He aquí el árbol de llamadas para el cálculo de *fibonacci*(4):



¿En qué orden se visitan los nodos del árbol? El orden de visita se indica en la siguiente figura con los números rodeados por un círculo.



---

360 Calcula F_{12} con ayuda de la función que hemos definido.

361 Dibuja el árbol de llamadas para *fibonacci*(5).

362 Modifica la función para que, cada vez que se la llame, muestre por pantalla un mensaje que diga «`Empieza cálculo de Fibonacci de` *n*», donde *n* es el valor del argumento, y para que, justo antes de acabar, muestre por pantalla «`Acaba cálculo de Fibonacci de` *n* `y devuelve el valor` *m*», donde *m* es el valor a devolver. A continuación, llama a la función para calcular el cuarto número de Fibonacci y analiza el texto que aparece por pantalla. Haz lo mismo para el décimo número de Fibonacci.

363 Puedes calcular recursivamente los números combinatorios sabiendo que, para $n \geq m$,

$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}$$

¿Programas eficientes o algoritmos eficientes?

Hemos presentado un programa recursivo para el cálculo de números de Fibonacci. Todo programa recursivo puede reescribirse con iteraciones. He aquí una versión iterativa:

fibonacci.py
```
def fibonacci_iterativo(n):
    if n == 1 or n == 2:
        f = 1
    else:
        f1 = f2 = 1
        for i in range(3, n+1):
            [f, f1, f2] = [f1 + f2, f2, f]
    return f
```

La función iterativa es *muchísimo* más rápida que la recursiva. La mayor rapidez no se debe a que haya menos llamadas a función, sino al propio algoritmo utilizado. El algoritmo recursivo que hemos diseñado tiene un *coste exponencial*, mientras que el iterativo tiene un *coste lineal*. ¿Que qué significa eso? Pues que el número de «pasos» del algoritmo lineal es directamente proporcional al valor de n, mientras que crece brutalmente en el caso del algoritmo recursivo: cada llamada a función genera (hasta) dos nuevas llamadas a función que, a su vez, generarán (hasta) otras dos cada una, y así sucesivamente. El número total de llamadas crece al mismo ritmo que 2^n... una función que crece vertiginosamente con n. Entonces, ¿un algoritmo iterativo es siempre preferible a uno recursivo? No. No siempre hay una diferencia de costes tan alta. En este caso, no obstante, podemos estar satisfechos del programa iterativo, al menos si lo comparamos con el recursivo. ¿Conviene usarlo siempre? No. El algoritmo iterativo no es el más eficiente de cuantos se conocen para el cálculo de números de Fibonacci. Hay una fórmula no recursiva que conduce a un algoritmo más eficiente:

$$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

Si defines una función que implemente ese cálculo, verás que es mucho más rápida que la función iterativa. Moraleja: la clave de un programa eficiente se encuentra (casi siempre) en diseñar (¡o encontrar en la literatura!) un algoritmo eficiente. Los libros de algorítmica son una excelente fuente de soluciones ya diseñadas por otros o, cuando menos, de buenas ideas aplicadas a otros problemas que nos ayudan a diseñar mejores soluciones para los nuestros. En un tema posterior estudiaremos la cuestión de la eficiencia de los algoritmos.

y que

$$\binom{n}{n} = \binom{n}{0} = 1.$$

Diseña un programa que, a partir de un valor n leído de teclado, muestre $\binom{n}{m}$ para m entre 0 y n. El programa llamará a una función *combinaciones* definida recursivamente.

▶ 364 El número de formas diferentes de dividir un conjunto de n números en k subconjuntos se denota con $\left\{ {n \atop k} \right\}$ y se puede definir recursivamente así:

$$\left\{ {n \atop k} \right\} = \left\{ {n-1 \atop k-1} \right\} + k \left\{ {n-1 \atop k} \right\}$$

El valor de $\left\{ {n \atop 1} \right\}$, al igual que el de $\left\{ {n \atop n} \right\}$, es 1. Diseña un programa que, a partir de un valor n leído de teclado, muestre $\left\{ {n \atop m} \right\}$ para m entre 0 y n. El programa llamará a una función *particiones* definida recursivamente.

6.8.4. El algoritmo de Euclides

Veamos otro ejemplo. Vamos a calcular el máximo común divisor (mcd) de dos números n y m por un procedimiento conocido como algoritmo de Euclides, un método que se conoce desde la

antigüedad y que se suele considerar el primer algoritmo propuesto por el hombre. El algoritmo dice así:

> «Calcula el resto de dividir el mayor de los dos números por el menor de ellos. Si el resto es cero, entonces el máximo común divisor es el menor de ambos números. Si el resto es distinto de cero, el máximo común divisor de *n* y *m* es el máximo común divisor de otro par de números: el formado por el menor de *n* y *m* y por dicho resto».

Resolvamos un ejemplo a mano. Calculemos el mcd de 500 y 218 paso a paso:

1) Queremos calcular el mcd de 500 y 218. Empezamos calculando el resto de dividir 500 entre 218: es 64. Como el resto no es cero, aún no hemos terminado. Hemos de calcular el mcd de 218 (el menor de 500 y 218) y 64 (el resto de la división).

2) Ahora queremos calcular el mcd de 218 y 64, pues ese valor será también la solución al problema original. El resto de dividir 218 entre 64 es 26, que no es cero. Hemos de calcular el mcd de 64 y 26.

3) Ahora queremos calcular el mcd de 64 y 26, pues ese valor será también la solución al problema original. El resto de dividir 64 entre 26 es 12, que no es cero. Hemos de calcular el mcd de 26 y 12.

4) Ahora queremos calcular el mcd de 26 y 12, pues ese valor será también la solución al problema original. El resto de dividir 26 entre 12 es 2, que no es cero. Hemos de calcular el mcd de 12 y 2.

5) Ahora queremos calcular el mcd de 12 y 2, pues ese valor será también la solución al problema original. El resto de dividir 12 entre 2 es 0. Por fin: el resto es nulo. El mcd de 12 y 2, que es el mcd de 26 y 12, que es el mcd de 64 y 26, que es el mcd de 218 y 64, que es el mcd de 500 y 218, es 2.

En el ejemplo desarrollado se hace explícito que una y otra vez resolvemos el mismo problema, solo que con datos diferentes. Si analizamos el algoritmo en términos de recursión encontramos que el caso base es aquel en el que el resto de la división es 0, y el caso general, cualquier otro.

Necesitaremos calcular el mínimo y el máximo de dos números, por lo que nos vendrá bien definir antes funciones que hagan esos cálculos[4]. Aquí tenemos el programa que soluciona recursivamente el problema:

```
mcd.py
def min(a, b):
    if a < b:
        return a
    else:
        return b

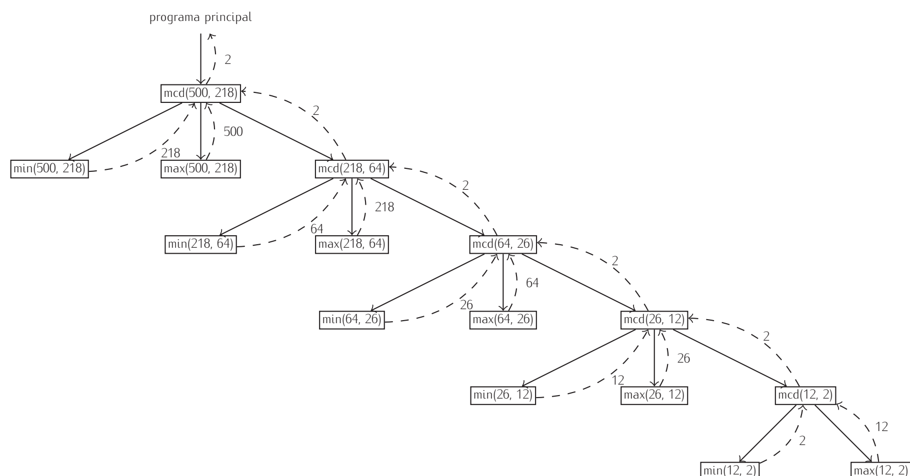
def max(a, b):
    if a > b:
        return a
    else:
        return b

def mcd(m, n):
    menor = min(m, n)
    mayor = max(m, n)
    resto = mayor % menor
    if resto == 0:
        return menor
```

[4] Fíjate: estamos aplicando la estrategia de diseño *ascendente*. Antes de saber qué haremos exactamente, ya estamos definiendo pequeñas funciones auxiliares que, seguro, nos interesará tener definidas.

```
    else:
        return mcd(menor, resto)
```

He aquí el árbol de llamadas para el cálculo de *mcd*(500,128):



▶ 365 Haz una traza de las llamadas a *mcd* para los números 1470 y 693.

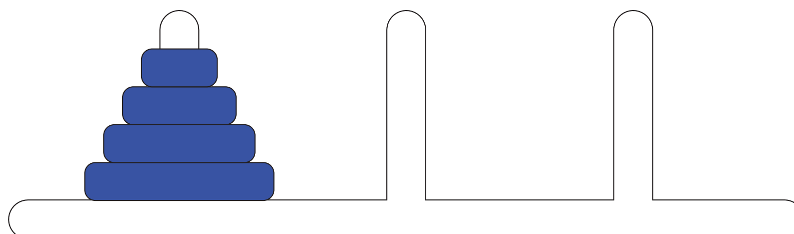
▶ 366 Haz una traza de las llamadas a *mcd* para los números 323 y 323.

▶ 367 En el apartado 6.6.4 presentamos el método de la bisección. Observa que, en el fondo, se trata de un método recursivo. Diseña una función que implemente el método de la bisección recursivamente.

6.8.5. Las torres de Hanoi

Cuenta la leyenda que en un templo de Hanoi, bajo la cúpula que señala el centro del mundo, hay una bandeja de bronce con tres largas agujas. Al crear el mundo, Dios colocó en una de ellas sesenta y cuatro discos de oro, cada uno de ellos más pequeño que el anterior hasta llegar al de la cima. Día y noche, incesantemente, los monjes transfieren discos de una aguja a otra siguiendo las inmutables leyes de Dios, que dicen que debe moverse cada vez el disco superior de los ensartados en una aguja a otra y que bajo él no puede haber un disco de menor radio. Cuando los sesenta y cuatro discos pasen de la primera aguja a otra, todos los creyentes se convertirán en polvo y el mundo desaparecerá con un estallido[5].

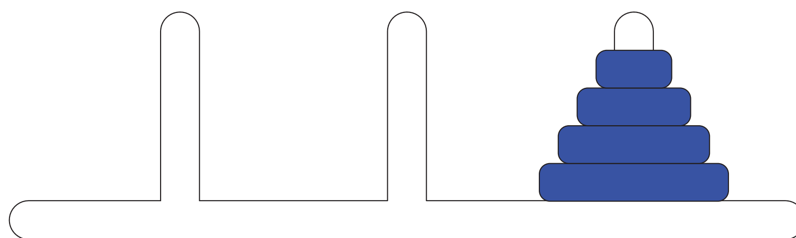
Nuestro objetivo es ayudar a los monjes con un ordenador. Entendamos bien el problema resolviendo a mano el juego para una torre de cuatro discos. La situación inicial es esta.



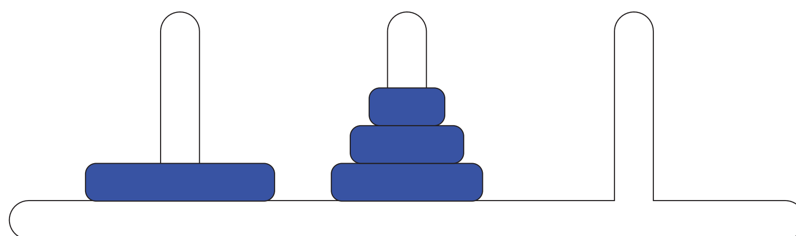
Y deseamos pasar a esta otra situación:

[5] La leyenda fue inventada por De Parville en 1884, en «Mathematical Recreations and Essays», un libro de pasatiempos matemáticos. La ambientación era diferente: el templo estaba en Benarés y el dios era Brahma.

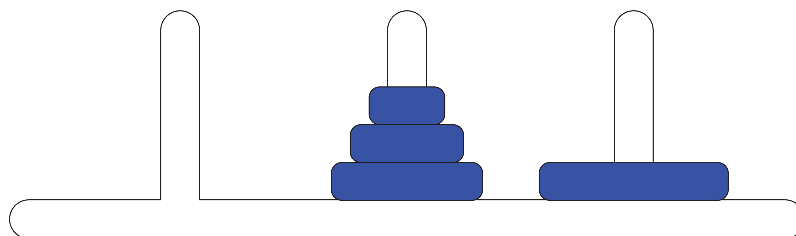
Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93
Índice



Aunque solo podemos tocar el disco superior de un montón, pensemos en el disco del fondo. Ese disco debe pasar de la primera aguja a la tercera, y para que eso sea posible, hemos de conseguir alcanzar esta configuración:



Solo en ese caso podemos pasar el disco más grande a la tercera aguja, es decir, alcanzar esta configuración:



Está claro que el disco más grande no se va a mover ya de esa aguja, pues es su destino final. ¿Cómo hemos pasado los tres discos superiores a la segunda aguja? Mmmm. Piensa que pasar una pila de tres discos de una aguja a otra no es más que el problema de las torres de Hanoi para una torre de tres discos. ¿Qué nos faltará por hacer? Mover la pila de tres discos de la segunda aguja a la tercera, y eso, nuevamente, es el problema de la torres de Hanoi para tres discos. ¿Ves cómo aparece la recursión? Resolver el problema de las torres de Hanoi con cuatro discos requiere:

- resolver el problema de las torres de Hanoi con tres discos, aunque pasándolos de la aguja inicial a la aguja libre;
- mover el cuarto disco de la aguja en que estaba inicialmente a la aguja de destino;
- y resolver el problema de las torres de Hanoi con los tres discos que están en la aguja central, que deben pasar a la aguja de destino.

La verdad es que falta cierta información en la solución que hemos esbozado: deberíamos indicar de qué aguja a qué aguja movemos los discos en cada paso. Reformulemos, pues, la solución y hagámosla general formulándola para n discos y llamando a las agujas inicial, libre y final (que originalmente son las agujas primera, segunda y tercera, respectivamente):

Resolver el problema de la torres de Hanoi con n discos que hemos de transferir de la aguja inicial a la aguja final requiere:

- *resolver el problema de las torres de Hanoi con* $n-1$ *discos de la aguja inicial a la aguja libre,*
- *mover el último disco de la aguja inicial a la aguja de destino,*

- *y resolver el problema de las torres de Hanoi con $n-1$ discos de la aguja libre a la aguja final.*

Hay un caso trivial o caso base: el problema de la torres de Hanoi para un solo disco (basta con mover el disco de la aguja en la que esté insertado a la aguja final). Ya tenemos, pues, los elementos necesarios para resolver recursivamente el problema.

¿Qué parámetros necesita nuestra función? Al menos necesita el número de discos que vamos a mover, la aguja origen y la aguja destino. Identificaremos cada aguja con un número. Esbocemos una primera solución:

```
hanoi.py
1  def resuelve_hanoi(n, inicial, final):
2      if n == 1:
3          print('Mover␣disco␣superior␣de␣aguja', inicial, 'a', final)
4      else:
5          # Determinar cuál es la aguja libre
6          if inicial != 1 and final != 1:
7              libre = 1
8          elif inicial != 2 and final != 2:
9              libre = 2
10         else:
11             libre = 3
12         # Primer subproblema: mover n-1 discos de inicial a libre
13         resuelve_hanoi(n-1, inicial, libre)
14         # Transferir el disco grande a su posición final
15         print('Mover␣disco␣superior␣de␣aguja', inicial, 'a', final)
16         # Segundo subproblema: mover n-1 discos de libre a final
17         resuelve_hanoi(n-1, libre, final)
```

Para resolver el problema con $n = 4$ invocaremos *resuelve_hanoi*(4,1,3).

Podemos presentar una versión más elegante que permite suprimir el bloque de líneas 5–11 añadiendo un tercer parámetro.

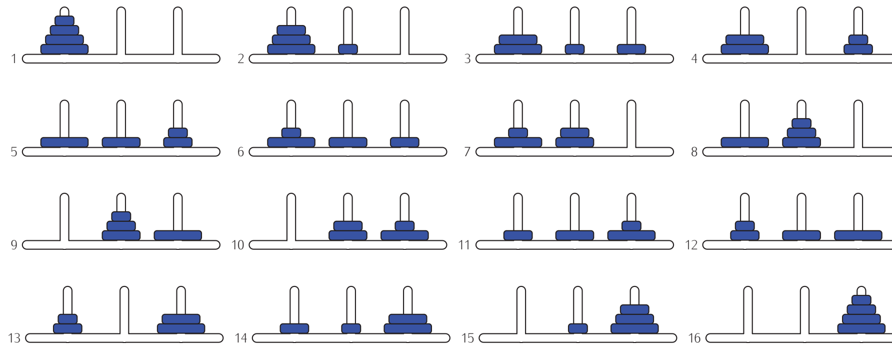
```
hanoi.py
def resuelve_hanoi(n, inicial, final, libre):
    if n == 1:
        print('Mover␣disco␣superior␣de␣aguja', inicial, 'a', final)
    else:
        resuelve_hanoi(n-1, inicial, libre, final)
        print('Mover␣disco␣superior␣de␣aguja', inicial, 'a', final)
        resuelve_hanoi(n-1, libre, final, inicial)

resuelve_hanoi(4,1,3,2)
```

El tercer parámetro se usa para «pasar» el dato de qué aguja está libre, y no tener que calcularla cada vez. Ahora, para resolver el problema con $n = 4$ invocaremos *resuelve_hanoi*(4, 1, 3, 2). Si lo hacemos, por pantalla aparece:

```
Mover disco superior de aguja 1 a 2
Mover disco superior de aguja 1 a 3
Mover disco superior de aguja 2 a 3
Mover disco superior de aguja 1 a 2
Mover disco superior de aguja 3 a 1
Mover disco superior de aguja 3 a 2
Mover disco superior de aguja 1 a 2
Mover disco superior de aguja 1 a 3
Mover disco superior de aguja 2 a 3
Mover disco superior de aguja 2 a 1
Mover disco superior de aguja 3 a 1
Mover disco superior de aguja 2 a 3
Mover disco superior de aguja 1 a 2
Mover disco superior de aguja 1 a 3
Mover disco superior de aguja 2 a 3
```

Ejecutemos las órdenes que imprime *resuelve_hanoi*:



▶ 368 Es hora de echar una manita a los monjes. Ellos han de resolver el problema con 64 discos. ¿Por qué no pruebas a ejecutar *resuelve_hanoi*(64, 1, 3, 2)?

▶ 369 ¿Cuántos movimientos son necesarios para resolver el problema de las torres de Hanoi con 1 disco, y con 2, y con 3, ...? Diseña una función *movimientos_hanoi* que reciba un número y devuelva el número de movimientos necesarios para resolver el problema de la torres de Hanoi con ese número de discos.

▶ 370 Implementa un programa gráfico con tortuga que muestre gráficamente la resolución del problema de las torres de Hanoi.

▶ 371 Dibuja el árbol de llamadas para *resuelve_hanoi*(4, 1, 3, 2).

6.8.6. Recursión indirecta

Las recursiones que hemos estudiado hasta el momento reciben el nombre de *recursiones directas*, pues una función se llama a sí misma. Es posible efectuar recursión indirectamente: una función puede llamar a otra quien, a su vez, acabe llamando a la primera.

Estudiemos un ejemplo sencillo, meramente ilustrativo de la idea y, la verdad, poco útil. Podemos decidir si un número natural es par o impar siguiendo los siguientes principios de *recursión indirecta*:

- un número n es par si $n - 1$ es impar,
- un número n es impar si $n - 1$ es par.
- 0 es par.

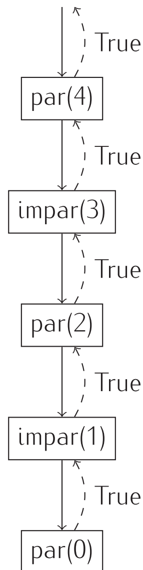
Podemos implementar en Python las funciones *par* e *impar* así:

```
par_impar.py
def par(n):
    if n == 0:
        return True
    else:
        return impar(n-1)

def impar(n):
    if n == 0:
        return False
    else:
        return par(n-1)
```

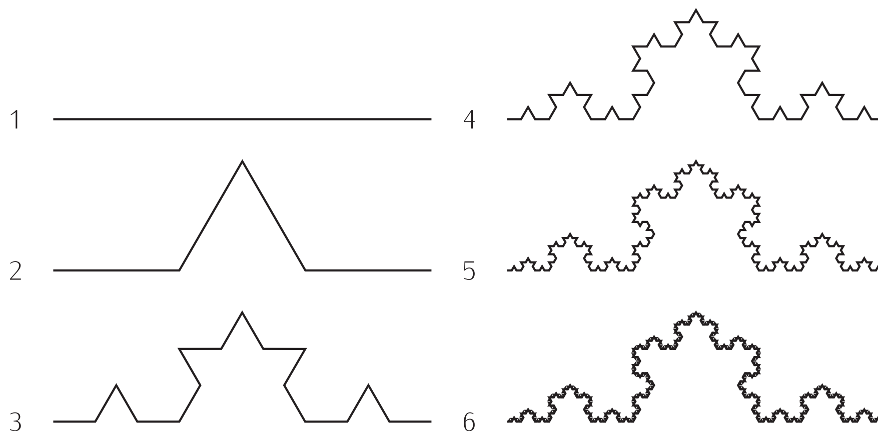
Fíjate en que el árbol de llamadas de *par*(4) alterna llamadas a *par* e *impar*:

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93



6.8.7. Gráficos fractales: copos de nieve de Von Koch

En 1904, Helge von Koch presentó en un trabajo científico una curiosa curva que da lugar a unos gráficos que hoy se conocen como copos de nieve de Von Koch. La curva de Von Koch se define recursivamente y es tanto más compleja cuanto más profunda es la recursión. He aquí algunos ejemplos de curvas de Von Koch con niveles de recursión crecientes:



El arte de la recursión

La recursión no es un concepto de exclusiva aplicación en matemáticas o programación. También el mundo de la literatura, el cine o el diseño han explotado la recursión. El libro de «Las mil y una noches», por ejemplo, es un relato que incluye relatos que, a su vez, incluyen relatos. Numerosas películas incluyen en su trama el rodaje o el visionado de otras películas: «Cantando bajo la lluvia», de Stanley Donen y Gene Kelly, «Nickelodeon», de Peter Bogdanovich, o «Vivir rodando», de Tom DiCillo, son películas en las que se filman otras películas; en «Angustia», de Bigas Luna, somos espectadores de una película en la que hay espectadores viendo otra película. Maurits Cornelius Escher es autor de numerosos grabados en los que está presente la recursión, si bien normalmente con *regresiones infinitas*. En su grabado «Manos dibujando», por ejemplo, una mano dibuja a otra que, a su vez, dibuja a la primera (una recursión indirecta).

El libro «Gödel, Escher, Bach: un Eterno y Grácil Bucle», de Douglas R. Hofstadter, es un apasionante ensayo sobre esta y otras cuestiones.

Los copos de nieve de Von Koch se forman combinando tres curvas de Von Koch que unen los vértices de un triángulo equilátero. Aquí tienes cuatro copos de nieve de Von Koch para niveles de recursión 0, 1, 2 y 3, respectivamente:



Estos gráficos reciben el nombre de «copos de nieve de Von Koch» porque recuerdan los diseños de cristalización del agua cuando forma copos de nieve.

Veamos cómo dibujar copos de nieve de Von Koch. Hemos de pensar en términos de la tortuga, que es la herramienta con la que dibujamos. Una curva de Koch viene descrita por dos elementos: la longitud y el nivel. Pensemos en la curva de Koch más sencilla: la de nivel cero y una longitud cualquiera. Esa curva es muy sencilla de trazar: es una simple línea de la longitud especificada. Vayamos ahora a por la curva de nivel 1 y longitud arbitraria. Esa curva se divide en 4 segmentos. El primero recorre una distancia de $l/3$, donde l es la longitud especificada; gira entonces 60 grados a la izquierda, avanza $l/3$, gira 120 grados a la derecha, avanza $l/3$, gira otros 60 grados a la izquierda y, finalmente, avanza una distancia $l/3$.

El procedimiento conduce a un método recursivo si reparamos en un importante detalle: cada uno de los 4 tramos que componen una curva de Koch de nivel 1 son, a su vez, curvas de Koch de nivel 0 (pues son simples líneas rectas). Esa es la clave. La curva de Koch de nivel n se forma con cuatro curvas de Koch de nivel $n-1$, cada una de las cuales se dibuja a partir de un giro apropiado a derecha o izquierda. Esta implementación refleja esa idea:

He aquí una implementación del algoritmo:

koch.py
```python
def koch(tortuga, longitud, nivel):
    if nivel == 0:
        tortuga.forward(longitud)
    else:
        koch(tortuga, longitud/3, nivel-1)
        tortuga.left(60)
        koch(tortuga, longitud/3, nivel-1)
        tortuga.right(120)
        koch(tortuga, longitud/3, nivel-1)
        tortuga.left(60)
        koch(tortuga, longitud/3, nivel-1)
```

Para tener un programa operativo necesitaremos un programa principal:

koch.py
```python
from turtle import Screen, Turtle

def koch(tortuga, longitud, nivel):
    if nivel == 0:
        tortuga.forward(longitud)
    else:
        koch(tortuga, longitud/3, nivel-1)
        tortuga.left(60)
        koch(tortuga, longitud/3, nivel-1)
        tortuga.right(120)
        koch(tortuga, longitud/3, nivel-1)
        tortuga.left(60)
        koch(tortuga, longitud/3, nivel-1)

pantalla = Screen()
pantalla.setup(500, 500)
pantalla.screensize(500, 500)
pantalla.setworldcoordinates(0, -250, 500, 250)
tortuga = Turtle()
```

```
tortuga.speed(9)
koch(tortuga, 400, 5)
pantalla.exitonclick()
```

El copo de Koch se obtiene uniendo tres curvas de Koch, cada una de las cuales está girada 120 grados a la derecha con respecto a la anterior. Esta función recibe como datos el tamaño de los segmentos principales y el nivel de recursión:

koch.py

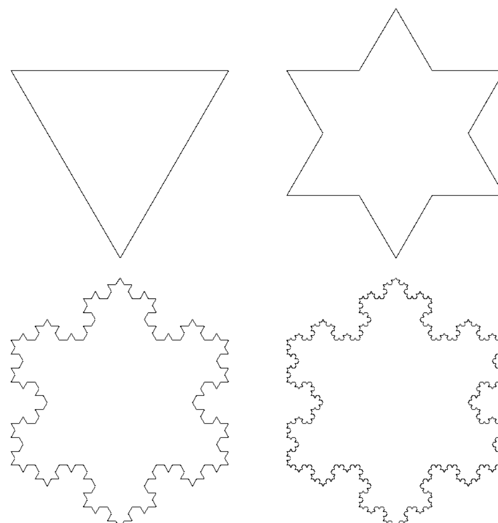
```
from turtle import Screen, Turtle

def koch(tortuga, longitud, nivel):
    if nivel == 0:
        tortuga.forward(longitud)
    else:
        koch(tortuga, longitud/3, nivel-1)
        tortuga.left(60)
        koch(tortuga, longitud/3, nivel-1)
        tortuga.right(120)
        koch(tortuga, longitud/3, nivel-1)
        tortuga.left(60)
        koch(tortuga, longitud/3, nivel-1)

def copo(tortuga, longitud, nivel):
    koch(tortuga, longitud, nivel)
    tortuga.right(120)
    koch(tortuga, longitud, nivel)
    tortuga.right(120)
    koch(tortuga, longitud, nivel)

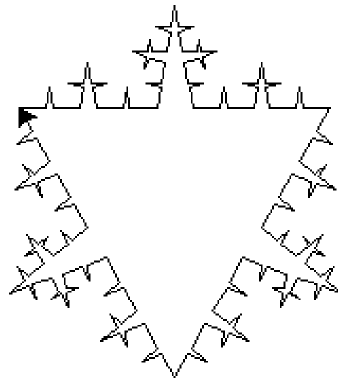
pantalla = Screen()
pantalla.setup(500, 500)
pantalla.screensize(500, 500)
pantalla.setworldcoordinates(0, -350, 500, 150)
tortuga = Turtle()
tortuga.speed(9)
copo(tortuga, 400, 3)
pantalla.exitonclick()
```

Aquí tienes el resultado de ejecutar la función con diferentes niveles de recursión (0, 1, 3 y 4, respectivamente):



▶ 372 Puedes jugar con los diferentes parámetros que determinan la curva de Koch para obtener infinidad de figuras diferentes. Basta con controlar los ángulos de giro y las longitudes apropiadamente:

```
def koch(tortuga, longitud, nivel):
    if nivel == 0:
        tortuga.forward(longitud)
    else:
        koch(tortuga, longitud/3, nivel-1)
        tortuga.left(80)
        koch(tortuga, longitud/3, nivel-1)
        tortuga.right(160)
        koch(tortuga, longitud/3, nivel-1)
        tortuga.left(80)
        koch(tortuga, longitud/3, nivel-1)
```



```
def koch(tortuga, longitud, nivel):
    if nivel == 0:
        tortuga.forward(longitud)
    else:
        koch(tortuga, longitud/3, nivel-1)
        tortuga.left(40)
        koch(tortuga, longitud/4, nivel-1)
        tortuga.right(160)
        koch(tortuga, longitud/6, nivel-1)
        tortuga.left(120)
        koch(tortuga, longitud/5, nivel-1)
```



Prueba a cambiar los diferentes parámetros y trata de predecir la figura que obtendrás en cada caso antes de ejecutar el programa.

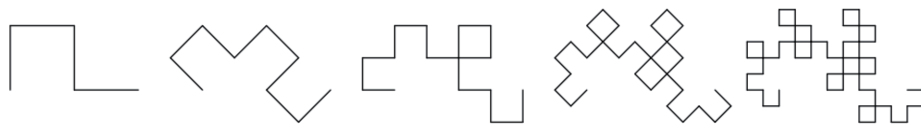
▶ 373 Estas dos funciones, mutuamente recursivas, definen otra estructura fractal interesante: la denominada curva dragón:

```
def dragon(tortuga, longitud, nivel):
```

```
    if nivel == 0:
        tortuga.forward(longitud)
    else:
        tortuga.right(45)
        dragon(tortuga, longitud/sqrt(2), nivel-1)
        tortuga.left(90)
        nogard(tortuga, longitud/sqrt(2), nivel-1)
        tortuga.right(45)

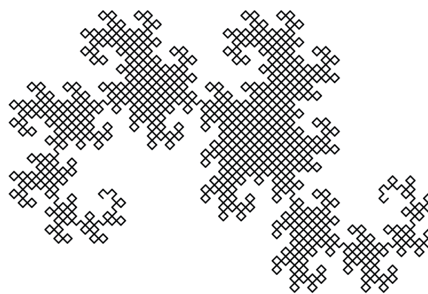
def nogard(tortuga, longitud, nivel):
    if nivel == 0:
        tortuga.forward(longitud)
    else:
        tortuga.left(45)
        dragon(tortuga, longitud/sqrt(2), nivel-1)
        tortuga.right(90)
        nogard(tortuga, longitud/sqrt(2), nivel-1)
        tortuga.left(45)
```

Trata de entender el principio de dibujo de este tipo de curva fractal. Aquí tienes las curvas dragón de niveles 2, 3, 4, 5 y 6.

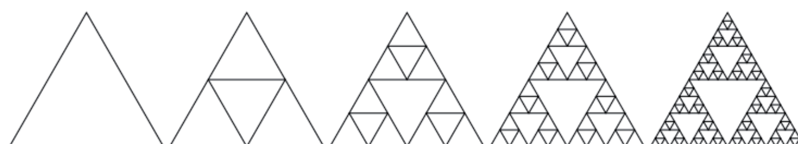


El perfil de la curvas dragón tiene una analogía con las dobleces de una hoja de papel. La curva dragón de nivel 0 es el perfil de una hoja de papel que no ha sido doblada. La de nivel 1 ha sido doblada una vez y desdoblada hasta que las partes dobladas forman ángulos de 90 grados. La curva dragón de nivel 2 es el perfil de una hoja doblada dos veces y desdoblada de forma que cada parte forme un ángulo de 90 grados con la siguiente.

Por cierto, ¿de dónde viene el nombre de «curva dragón»? Del aspecto que presenta en niveles «grandes». Aquí tienes la curva dragón de nivel 11:

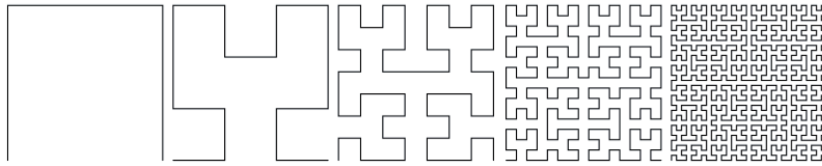


▶ 374 Otra figura recursiva que es todo un clásico es la criba o triángulo de Sierpinski. En cada nivel de recursión se divide cada uno de los triángulos del nivel anterior en tres nuevos triángulos. Esta figura muestra los triángulos de Sierpinski para niveles de recursión de 0 a 4:

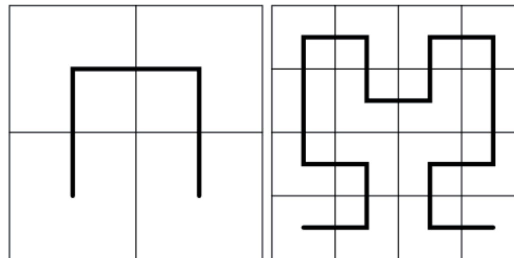


Diseña un programa que dibuje triángulos de Sierpinski para un nivel de recursión dado.

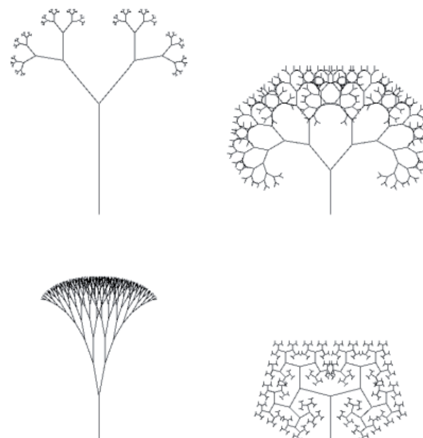
▶ 375 Otra curva fractal de interés es la denominada «curva de relleno del espacio de Hilbert». Esta figura te muestra dicha curva con niveles de recursión 0, 1, 2, 3 y 4:



Diseña un programa capaz de dibujar curvas de relleno del espacio de Hilbert dado el nivel de recursión deseado. Estas figuras te pueden ser de ayuda para descubrir el procedimiento de cálculo que has de programar:



▶ 376 Una curiosa aplicación de la recursión es la generación de paisajes por ordenador. Las montañas, por ejemplo, se dibujan con modelos recursivos: los denominados fractales (las curvas de Von Koch, entre otras, son fractales). Los árboles pueden generarse también con procedimientos recursivos. Estas imágenes, por ejemplo, muestran «esqueletos» de árboles generados con Python:



Todos han sido generados con una misma función recursiva, pero usando diferentes argumentos. Te vamos a describir el principio básico de generación de estos árboles, pero has de ser tú mismo quien diseñe una función recursiva capaz de efectuar este tipo de dibujos. Vamos con el método. El usuario nos proporciona los siguientes datos:

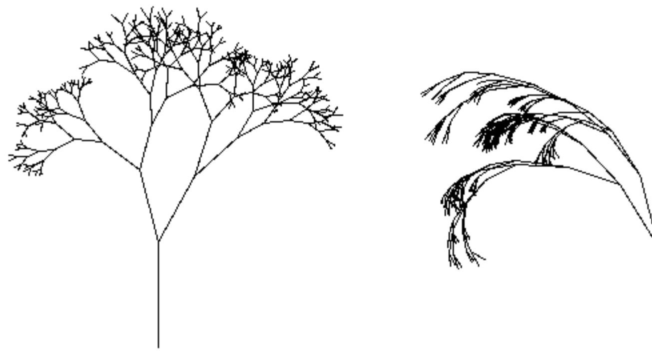
- Los puntos en los que empieza y acaba el tronco.
- El ángulo α que forma la rama que parte a mano derecha del tronco con el propio tronco. La rama que parte a mano izquierda lo hace con un ángulo $-\alpha$.
- La proporción (en tanto por uno) del tamaño de las ramas con respecto al tronco.
- El nivel de recursión deseado.

La recursión tiene lugar cuando consideramos que cada una de las dos ramas es un nuevo tronco.

Por cierto, los árboles ganan bastante si en los primeros niveles de recursión usas un color anaranjado o marrón y en los últimos usas un color verde.

▶ 377 Los árboles que hemos generado en el ejercicio anterior parecen un tanto artificiales por ser tan regulares y simétricos. Introducir el azar en su diseño los hará parecer más naturales. Modifica la función del ejercicio anterior para que tanto el ángulo como la proporción rama/tronco se escojan aleatoriamente (dentro de ciertos márgenes).

Aquí tienes un par de ejemplos. El árbol de la izquierda sí parece bastante real y el de la derecha parece mecido por el viento (bueno, ¡más bien por un huracán!).



6.9. Módulos

Las funciones ayudan a hacer más legibles tus programas y a evitar que escribas una y otra vez los mismos cálculos en *un* mismo programa. Sin embargo, cuando escribas *varios* programas, posiblemente descubrirás que acabas escribiendo la misma función en cada programa… a menos que escribas tus propios módulos.

Los módulos son colecciones de funciones que puedes utilizar desde tus programas. Conviene que las funciones se agrupen en módulos según su ámbito de aplicación.

La distribución estándar de Python nos ofrece gran número de módulos predefinidos. Cada módulo agrupa las funciones de un ámbito de aplicación. Las funciones matemáticas se agrupan en el módulo *math*; las que analizan documentos HTML (el lenguaje de marcas del World Wide Web) en *htmllib*; las que generan números al azar, en *random*; las que trabajan con fechas de calendario, en *calendar*; las que permiten montar un cliente propio de FTP (un protocolo de intercambio de ficheros en redes de ordenadores), en *ftplib*… Como ves, Python tiene una gran colección de módulos predefinidos. Conocer aquellos que guardan relación con las áreas de trabajo para las que vas a desarrollar programas te convertirá en un programador más eficiente: ¿para qué volver a escribir funciones que ya han sido *escritas por otros*?[6]

En esta sección aprenderemos a crear y usar nuestros propios módulos. Así, podremos reutilizar funciones que ya hemos escrito al solucionar un problema de programación: ¿para qué volver a escribir funciones que ya han sido *escritas por nosotros mismos*?[7]

6.9.1. Un módulo muy sencillo: mínimo y máximo

Empezaremos creando un módulo con las funciones *min* y *max* que definimos en un ejemplo anterior. Llamaremos al módulo *minmax*, así que deberemos crear un fichero de texto llamado **minmax.py**. El sufijo o *extensión* **py** sirve para indicar que el fichero contiene código Python. Este es el contenido del fichero:

```
minmax.py
def min(a, b):
```

[6]Bueno, si estás aprendiendo a programar, sí tiene algún sentido.

[7]Bueno, si estás aprendiendo a programar, sí tiene algún sentido.

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

```
    if a < b:
        return a
    else:
        return b

def max(a, b):
    if a > b:
        return a
    else:
        return b
```

En cualquier programa donde deseemos utilizar las funciones *min* y *max* bastará con incluir antes la siguiente línea:

mi_programa.py

```
from minmax import min, max
```

Observa que escribimos «from minmax», y no «from minmax.py»: la extensión del fichero no forma parte del nombre del módulo.

▶ 378 Construye un módulo llamado *dni* que incluya las funciones de cálculo de la letra del DNI.

Usa el módulo desde un programa que pida al usuario su número de DNI y su letra. Si el usuario mete un número y letra de DNI correctos, el programa emitirá el mensaje «Bienvenido». En caso contrario dirá «Ha cometido ud. un error».

minmax.py y minmax.pyc

Cuando importas por primera vez el módulo minmax.py, Python crea automáticamente un fichero llamado minmax.pyc. Ese fichero contiene una versión de tu módulo más fácil de cargar en memoria para Python, pero absolutamente ilegible para las personas: está codificado en lo que llamamos «formato binario». Python pretende acelerar así la carga de módulos que usas en tus programas, pero sin obligarte a ti a gestionar los ficheros pyc.

Si borras el fichero minmax.pyc, no pasará nada grave: sencillamente, Python lo volverá a crear cuando cargues nuevamente el módulo minmax.py desde un programa cualquiera. Si modificas el contenido de minmax.py, Python regenera automáticamente el fichero minmax.pyc para que siempre esté «sincronizado» con minmax.py.

6.9.2. Un módulo más interesante: gravedad

En un módulo no solo puede haber funciones: también puedes definir variables cuyo valor debe estar predefinido. Por ejemplo, el módulo matemático (*math*) incluye constantes como *pi* o *e* que almacenan (sendas aproximaciones a) el valor de π y e, respectivamente. Para definir una variable en un módulo basta con incluir una asignación en el fichero de texto.

Vamos con un nuevo ejemplo: un módulo con funciones y constantes físicas relacionadas con la gravitación. Pero antes, un pequeño repaso de física.

La fuerza (en Newtons) con que se atraen dos cuerpos de masa M y m (en kilogramos) separados una distancia r (en metros) es

$$F = G\frac{Mm}{r^2},$$

donde G es la denominada constante de gravitación universal. G vale, aproximadamente, $6.67 \times 10^{-11}\,\mathrm{N\,m^2\,kg^{-2}}$. Por otra parte, la velocidad de escape de un planeta para un cuerpo cualquiera es

$$v_e = \sqrt{\frac{2GM}{R}},$$

Probando los módulos

Una vez has escrito un módulo es buena práctica probar que funciona correctamente. Puedes crear un programa que utilice a tu módulo en muchas circunstancias diferentes para ver que proporciona los resultados correctos. En ese caso tendrás dos ficheros de texto: el fichero que corresponde al módulo en sí y el que contiene el programa de pruebas. Python te permite que el contenido de ambos ficheros resida en uno solo: el del módulo.

El siguiente texto reside en un único fichero (`minmax.py`):

minmax.py
```
def min(a, b):
    if a < b:
        return a
    else:
        return b

def max(a, b):
    if a > b:
        return a
    else:
        return b

if __name__ == '__main__':
    print('El máximo de 3 y 10 es',  max(3,10))
    print('El máximo de 3 y -10 es', max(3,-10))
    print('El mínimo de 3 y 10 es',  min(3,10))
    print('El mínimo de 3 y -10 es', min(3,-10))
```

Si lo que hacemos es *importar* el módulo *minmax* desde otro fichero, así:

programa.py
```
from minmax import min, max
```

la variable *__name__* vale `'minmax'`, que es como se llama el módulo. De este modo podemos saber si el código del fichero se está ejecutando o importando. Pues bien, el truco está en ejecutar la batería de pruebas solo cuando el fichero se está ejecutando.

Máximo y mínimo

Ya te hemos comentado que Python trae muchas utilidades «de fábrica». Las funciones de cálculo del máximo y el mínimo parecen muy útiles, así que sería de extrañar que no estuvieran predefinidas. Pues bien, lo están: la función *max* calcula el máximo y *min* el mínimo. Fíjate:

```
>>> print(max(1,3))
3
>>> print(min(3, 2, 8, 10, 7))
2
```

Las funciones *max* y *min* funcionan con cualquier número de argumentos mayor que cero. ¿Recuerdas los ejercicios en que te pedíamos calcular el mayor (o menor) de 5 números? ¡Entonces sí que te hubiera venido bien saber que existían *max* (o *min*)!

Estas funciones también trabajan con listas:

```
>>> a = [10, 2, 38]
>>> print(max(a))
38
>>> print(min(a))
2
```

Lo cierto es que *max* y *min* funcionan con cualquier tipo de secuencia. Una curiosidad: ¿qué crees que devolverá *max*(`'una cadena'`)? ¿Y *min*(`'una cadena'`)?

donde M es la masa del planeta (en kilogramos) y R su radio (en metros).

Nuestro módulo, al que denominaremos *gravedad*, exportará unas cuantas constantes:

- *G*: la constante de gravitación universal.
- *MTierra*: la masa de la Tierra.
- *RTierra*: el radio de la Tierra.
- *veTierra*: la velocidad de escape de la Tierra.
- *MLuna*: la masa de la Luna.
- *RLuna*: el radio de la Luna.
- *veLuna*: la velocidad de escape de la Luna.

Por cierto, la masa de la Tierra es de 5.97×10^{24} kilogramos y su radio es de 6.37×10^{6} metros; y la masa de la Luna es de 7.35×10^{22} kilogramos y su radio es de 1.74×10^{6} metros.

Por otra parte, el módulo definirá las siguientes funciones:

- *fuerzaGravitatoria*: recibe la masa de dos cuerpos (en kilogramos) y la distancia que los separa (en metros) y devuelve la fuerza gravitatoria que experimentan (en Newtons).
- *distancia*: recibe la masa de dos cuerpos (en kilogramos) y la fuerza gravitatoria que experimentan por efecto mutuo (en Newtons) y devuelve la distancia que los separa (en metros).
- *velocidadEscape*: recibe la masa (en kilogramos) y el radio (en metros) de un planeta y devuelve la velocidad (en metros por segundo) que permite a un cuerpo cualquiera escapar de la órbita del planeta.

He aquí (una primera versión de) el contenido del fichero `gravedad.py` (recuerda que el fichero debe finalizar con la extensión `py`):

```
gravedad.py
1  from math import sqrt
2
3  G = 6.67e-11
4  MTierra = 5.97e24
5  RTierra = 6.37e6
6  MLuna = 7.35e22
7  RLuna = 1.74e6
8
9  def fuerzaGravitatoria(M, m, r):
10     return G * M * m / r**2
11
12 def distancia(M, m, F):
13     return sqrt( G * M * m / F )
14
15 def velocidadEscape(M, R):
16     return sqrt( 2 * G * M / R )
17
18 veTierra = velocidadEscape(MTierra, RTierra)
19 veLuna = velocidadEscape(MLuna, RLuna)
```

Observa que las variables *veTierra* y *veLuna* se han definido al final (líneas 18 y 19). Lo hemos hecho así para poder aprovechar la función *velocidadEscape*, que ha de estar definida antes de ser usada (líneas 15–16). Por otra parte, el módulo utiliza una función (*sqrt*) del módulo matemático, así que empieza importándola (línea 1).

Acabaremos mostrando un ejemplo de uso del módulo *gravedad* desde un programa (que estará escrito en otro fichero de texto):

```
escapes.py
from gravedad import velocidadEscape, veTierra
```

```
print('La velocidad de escape de Plutón es', end='')
print('de', velocidadEscape(1.29e22, 1.16e6), 'm/s.')
print('La de la Tierra es de', veTierra, 'm/s.')
```

6.10. Documentación del código

Ya empezamos a crear programas de cierta entidad. ¡Y solo estamos aprendiendo a programar! Cuando trabajes con programas del «mundo real», verás que estos se dividen en numerosos módulos y, generalmente, cada uno de ellos define muchas funciones y constantes. Esos programas, por regla general, no son obra de un solo programador, sino de un equipo de programadores. Muchas veces, el autor o autores de un módulo necesitan consultar módulos escritos por otros autores, o a un programador se le puede encargar que siga desarrollando un módulo de otros programadores, o que modifique un módulo que él mismo escribió hace mucho tiempo. Es vital, pues, que los programas sean *legibles* y estén bien *documentados*.

Hemos de acostumbrarnos a documentar el código. Nuestro módulo estará incompleto sin una buena documentación:

gravedad.py

```
#------------------------------------------------------------
# Módulo: gravedad
#
# Propósito: proporciona algunas constantes y funciones sobre física gravitatoria.
#
# Autor/es: Isaac Pérez González y Alberto Pérez López
#------------------------------------------------------------
# Constantes exportadas:
# G: Constante de gravitación universal.
# MTierra: Masa de la Tierra (en kilos).
# RTierra: Radio de la Tierra (en metros).
# MLuna: Masa de la Luna (en kilos).
# RLuna: Radio de la Luna (en metros).
#
# Funciones exportadas:
# fuerzaGravitatoria: calcula la fuerza gravitatoria existente entre dos cuerpos.
# entradas:
# M: masa de un cuerpo (en kg).
# m: masa del otro cuerpo (en kg).
# r: distancia entre ellos (en metros).
# salida:
# fuerza (en Newtons).
#
# distancia: calcula la distancia que separa dos cuerpos atraídos por una fuerza
# gravitatoria determinada.
# entradas:
# M: masa de un cuerpo (en kg).
# m: masa del otro cuerpo (en kg).
# F: fuerza gravitatoria experimentada (en m).
# salida:
# distancia (en metros).
#
# velocidadEscape: calcula la velocidad necesaria para escapar de la atracción
# gravitatoria de un cuerpo esférico.
# entradas:
# M: masa del cuerpo (en kg).
# R: radio del cuerpo (en metros).
# salida:
# velocidad (en metros por segundo).
#------------------------------------------------------------
```

```
# Historia:
# * Creado el 13/11/2001 por Isaac
# * Modificado el 15/11/2001 por Alberto:
# - se incluyen las constantes MLuna y RLuna
# - se añade la función velocidadEscape
#-----------------------------------------------------------------
from math import sqrt

G = 6.67e-11
MTierra = 5.97e24
RTierra = 6.37e6
MLuna = 7.35e22
RLuna = 1.74e6

def fuerzaGravitatoria(M, m, r):
    return G * M * m / r**2

def distancia(M, m, F):
    return sqrt( G * M * m / F )

def velocidadEscape(M, R):
    return sqrt( 2 * G * M / R )

veTierra = velocidadEscape(MTierra, RTierra)
veLuna = velocidadEscape(MLuna, RLuna)
```

De acuerdo, el módulo es ahora mucho más largo, pero está bien documentado. Cualquiera puede averiguar su utilidad con solo leer la cabecera.

Ándate con ojo: no todos los comentarios son interesantes. Este, por ejemplo, es absurdo:

```
# Devuelve el producto de G por M y m dividido por r al cuadrado.
return G * M * m / r ** 2
```

Lo que dice ese comentario es una obviedad. En este caso, el comentario no ayuda a entender nada que no esté ya dicho en la propia sentencia. Más que ayudar, distrae al lector. La práctica te hará ir mejorando el estilo de tus comentarios y te ayudará a decidir cuándo convienen y cuándo son un estorbo.

▶ 379 Diseña un módulo que agrupe las funciones relacionadas con hipotecas de los ejercicios 304–307. Documenta adecuadamente el módulo.

6.10.1. Otro módulo: cálculo vectorial

Vamos a desarrollar ahora un módulo para cálculo vectorial en tres dimensiones. Un vector tridimensional (x, y, z) se representará mediante una lista con tres elementos numéricos: `[x,y,z]`. Nuestro módulo suministrará funciones y constantes útiles para el cálculo con este tipo de datos.

Empezaremos definiendo una a una las funciones y constantes que ofrecerá nuestro módulo. Después mostraremos el módulo completo.

Definamos una función que sume dos vectores. Primero hemos de tener claro cómo se define matemáticamente la suma de vectores: $(x, y, z) + (x', y', z') = (x + x', y + y', z + z')$. Llamaremos *vSuma* a la operación de suma de vectores:

```
def vSuma(u, v):
    return [ u[0] + v[0], u[1] + v[1], u[2] + v[2] ]
```

La longitud de un vector (x, y, z) es $\sqrt{x^2 + y^2 + z^2}$. Definamos una función *vLongitud*:

```
def vLongitud(v):
```

```
    return sqrt(v[0]**2 + v[1]**2 + v[2]**2)
```

Recuerda que antes deberemos importar *sqrt* del módulo *math*.

El producto escalar de dos vectores (x, y, z) y (x', y', z') es una cantidad escalar igual a $xx' + yy' + zz'$:

```
def vProductoEscalar(u, v):
    return u[0]*v[0] + u[1]*v[1] + u[2]*v[2]
```

Dos vectores son perpendiculares si su producto escalar es cero. Construyamos una función que devuelva True cuando dos vectores son perpendiculares y False en caso contrario:

```
def vSonPerpendiculares(u, v):
    return vProductoEscalar(u, v) == 0
```

El producto vectorial de dos vectores (x, y, z) y (x', y', z') se define como un nuevo vector $(yz' - zy', zx' - xz', xy' - yx')$:

```
def vProductoVectorial(u, v):
    resultado_x = u[1]*v[2] - u[2]*v[1]
    resultado_y = u[2]*v[0] - u[0]*v[2]
    resultado_z = u[0]*v[1] - u[1]*v[0]
    return [resultado_x, resultado_y, resultado_z]
```

Para facilitar la introducción de vectores, vamos a definir una función *vLeeVector* que lea de teclado las tres componentes de un vector:

```
def vLeeVector():
    x = float(input('Componente x:'))
    y = float(input('Componente y:'))
    z = float(input('Componente z:'))
    return [x, y, z]
```

Y para facilitar la impresión de vectores, definiremos un procedimiento que muestra un vector por pantalla siguiendo la notación habitual en matemáticas (con paréntesis en lugar de corchetes):

```
def vMuestraVector(v):
    print('({0}, {1}, {2})'.format(v[0], v[1], v[2]))
```

Los vectores $\mathbf{i} = (1, 0, 0)$, $\mathbf{j} = (0, 1, 0)$ y $\mathbf{k} = (0, 0, 1)$ se definirán en nuestro módulo como las variables *vI*, *vJ* y *vK*, respectivamente.

```
vI = [1, 0, 0]
vJ = [0, 1, 0]
vK = [0, 0, 1]
```

Bueno, es hora de juntarlo todo en un módulo. En un fichero llamado *vectores.py* tecleamos el siguiente texto:

vectores.py

```
#----------------------------------------------------------
# Módulo vectores
#----------------------------------------------------------
# Proporciona constantes y funciones para el cálculo vectorial en 3 dimensiones.
#----------------------------------------------------------
# Constantes que exporta:
# vI, vJ, vK: vectores unidad
#
```

```
# Funciones que exporta:
# vLeeVector:
# sin parámetros
# devuelve un vector leído de teclado que se pide al usuario
#
# vMuestraVector(v):
# muestra por pantalla el vector v con la notación(x,y,z)
# no devuelve nada
#
# vLongitud()v):
# devuelve la longitud del vector v
#
# vSuma(u, v):
# devuelve el vector resultante de sumar u y v
#
# vProductoEscalar(u, v):
# devuelve el escalar resultante del producto escalar de u por v
#
# vProductorVectorial()u, v):
# devuelve el vector resultante del producto vectorial de u por v
#
# vSonPerpendiculares(u, v):
# devuelve cierto si u y v son perpendiculares, y falso en caso contrario
#-------------------------------------------------------------

# Funciones matemáticas utilizadas

from math import sqrt

# Constantes

vI = [1, 0, 0]
vJ = [0, 1, 0]
vK = [0, 0, 1]

# Funciones de entrada/salida

def vLeeVector():
    x = float(input('Componente␣x:'))
    y = float(input('Componente␣y:'))
    z = float(input('Componente␣z:'))
    return [x, y, z]

def vMuestraVector(v):
    print('({0},␣{1},␣{2})'.format(v[0], v[1], v[2]))

# Funciones de cálculo

def vLongitud(v):
    return sqrt(v[0]**2 + v[1]**2 + v[2]**2)

def vSuma(u, v):
    return [ u[0] + v[0], u[1] + v[1], u[2] + v[2] ]

def vProductoEscalar(u, v):
    return u[0]*v[0] + u[1]*v[1] + u[2]*v[2]

def vProductoVectorial(u, v):
    resultado_x = u[1]*v[2] - u[2]*v[1]
    resultado_y = u[2]*v[0] - u[0]*v[2]
    resultado_z = u[0]*v[1] - u[1]*v[0]
```

```
    return [resultado_x, resultado_y, resultado_z]

# Predicados

def vSonPerpendiculares(u, v):
    return vProductoEscalar(u, v) == 0
```

▶ 380 Diseña un módulo similar al anterior pero que permita efectuar cálculos con vectores n-dimensionales, donde n es un valor arbitrario. Las funciones que debes definir son:

- *vLeeVector*: Pide el valor de n y a continuación lee los n componentes del vector. El resultado devuelto es la lista de los componentes.
- *vMuestraVector*: Muestra por pantalla el vector en la notación $(v_1, v_2, \ldots, v_n)$.
- *vLongitud*: devuelve la longitud del vector, que es

$$\sqrt{\sum_{i=1}^{n} v_i^2}$$

- *vSuma*: Devuelve la suma de dos vectores. Los dos vectores deben tener la misma dimensión. Si no la tienen, *vSuma* devolverá el valor **None**.
- *vProductoEscalar*: Devuelve el producto escalar de dos vectores. Los dos vectores deben tener la misma dimensión. Si no la tienen, la función devolverá el valor **None**.

▶ 381 Diseña un módulo que facilite el trabajo con conjuntos. Recuerda que un conjunto es una lista en la que no hay elementos repetidos. Deberás implementar las siguientes funciones:

- *lista_a_conjunto*(*lista*): Devuelve un conjunto con los mismos elementos que hay en *lista*, pero sin repeticiones. (Ejemplo: *lista_a_conjunto*([1,1,3,2,3]) devolverá la lista [1, 2, 3], aunque también se acepta como equivalente cualquier permutación de esos mismos elementos, como [3,1,2] o [3,2,1]).
- *unión*(*A*, *B*): devuelve el conjunto resultante de unir los conjuntos A y B.
- *intersección*(*A*, *B*): devuelve el conjunto cuyos elementos pertenecen a A y a B.
- *diferencia*(*A*, *B*): devuelve el conjunto de elementos que pertenecen a A y no a B.
- *iguales*(*A*, *B*): devuelve cierto si A y B tienen los mismos elementos y falso en caso contrario. (Nota: ten en cuenta que los conjuntos representados por las listas [1, 3, 2] y [2, 1, 3] son iguales).

6.10.2. Un módulo para trabajar con polinomios

Supón que deseamos trabajar con polinomios, es decir, con funciones de la forma

$$f(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \cdots + a_nx^n.$$

Nos interesará poder operar con polinomios. Diseñaremos un módulo que permita:

- Mostrar por pantalla los polinomios en una notación similar a la matemática.
- Evaluar un polinomio para un valor dado de x.
- Obtener el polinomio que resulta de sumar otros dos.

- Obtener el polinomio que resulta de restar un polinomio a otro.
- Obtener el polinomio que resulta de multiplicar dos polinomios.

Empezaremos por decidir una representación para los polinomios. Un polinomio de orden n es una lista de $n+1$ elementos: los $n+1$ coeficientes del polinomio. El polinomio

$$1 + 2x + 4x^2 - 5x^3 + 6x^5$$

es de orden 5, así que se representará con una lista de 6 elementos: `[1, 2, 4, -5, 0, 6]`.

Ahora que hemos decidido la representación que usaremos, hagamos un procedimiento que muestre por pantalla un polinomio en un formato «agradable» y no como una lista de números:

```
def muestra(p):
    print(p[0], end='')
    for i in range(1, len(p)):
        print('+', p[i], 'x␣**', i, end='')
    print()
```

Diseñemos la función que evalúe un polinomio p para un valor dado de x:

```
def evalúa(p, x):
    suma = 0
    for i in range(len(p)):
        suma = suma + p[i] * x**i
    return suma
```

Vamos a por la función que suma dos polinomios. Antes de empezar, entendamos qué hay que hacer. Supongamos que hemos de sumar los polinomios $a_0+a_1x+\cdots+a_nx^n$ y $b_0+b_1x+\cdots+b_nx^n$. Fácil: la solución es un polinomio $c_0 + c_1x + \cdots + c_nx^n$ donde $c_i = a_i + b_i$, para i entre 0 y n. Bueno, este caso era particularmente fácil porque ambos polinomios eran del mismo orden. Si los polinomios sumados son de órdenes distintos deberemos llevar más cuidado.

Lo que no va a funcionar es el operador `+`, pues al trabajar con listas efectúa una concatenación. Es decir, si concatenamos las listas `[1, 2, 3]` y `[1, 0, -1]`, que representan polinomios de orden 2, obtenemos un polinomio de orden 5 (el representado por la lista `[1, 2, 3, 1, 0, -1]`), y eso es incorrecto.

Vamos con una propuesta de función *suma*:

```
def suma(a, b):
    # creamos un polinomio nulo de orden igual al de mayor orden
    c = [0] * max(len(a), len(b))
    # sumamos los coeficientes hasta el orden menor
    for i in range(min(len(a), len(b))):
        c[i] = a[i] + b[i]
    # y ahora copiamos el resto de coeficientes del polinomio de mayor orden.
    if len(a) > len(b):
        for i in range(len(b), len(c)):
            c[i] = a[i]
    else:
        for i in range(len(a), len(c)):
            c[i] = b[i]
    # y devolvemos el polinomio c
    return c
```

▶ 382 ¿Es correcta esta otra versión de la función *suma*?

```
def suma(a, b):
    c = []
    m = min(len(a), len(b))
    for i in range(m):
```

```
        c.append(a[i] + b[i])
    c = c + a[m:] + b[m:]
    return c
```

Ya casi está. Hay un pequeño detalle: imagina que sumamos los polinomios representados por [1, 2, 3] y [1, 2, -3]. El polinomio resultante es [2, 4, 0]. Bien, pero ese polinomio es un poco «anormal»: parece de orden 2, pero en realidad es de orden 1, ya que el último coeficiente, el que afecta a x^2 es nulo. Diseñemos una función que «normalice» los polinomios eliminando los coeficientes nulos a la derecha del todo:

```
def normaliza(p):
    while len(p) > 0 and p[-1] == 0:
        del p[-1]
```

Nuestra función *suma* (y cualquier otra que opere con polinomios) deberá asegurarse de que devuelve un polinomio normalizado:

```
def suma(a, b):
    # creamos un polinomio nulo de orden igual al de mayor orden
    c = [0] * max(len(a), len(b))
    # sumamos los coeficientes hasta el orden menor
    for i in range(min(len(a), len(b))):
        c[i] = a[i] + b[i]
    # y ahora copiamos el resto de coeficientes del polinomio de mayor orden.
    if len(a) > len(b):
        for i in range(len(b), len(c)):
            c[i] = a[i]
    else:
        for i in range(len(a), len(c)):
            c[i] = b[i]
    # normalizamos y devolvemos el polinomio c
    normaliza(c)
    return c
```

La función que resta un polinomio de otro te la dejamos como ejercicio. Vamos con el producto de polinomios, que es una función bastante más complicada. Si multiplicamos dos polinomios a y b de órdenes n y m, respectivamente, el polinomio resultante c es de orden $n+m$. El coeficiente de orden c_i se obtiene así:

$$c_i = \sum_{j=0}^{i} a_j b_{i-j}.$$

Vamos con la función:

```
def multiplica(a, b):
    orden = len(a) + len(b) - 2
    c = [0] * (orden + 1)
    for i in range(orden+1):
        suma = 0
        for j in range(i+1):
            suma += a[j] * b[i-j]
        c[i] = suma
    return c
```

Encárgate tú ahora de unir las funciones desarrolladas en un módulo llamado *polinomios*.

▶ 383 Diseña el siguiente programa que usa el módulo *polinomios* y, si te parece conveniente, enriquece dicho módulo con nuevas funciones útiles para el manejo de polinomios. El programa presentará al usuario este menú:

```
1) Leer polinomio p
2) Mostrar polinomio p
3) Leer polinomio q
4) Mostrar polinomio q
5) Sumar polinomios p y q
6) Restar p de q
7) Restar q de p
8) Multiplicar p por q
9) FIN DE PROGRAMA
```

6.10.3. Un módulo con utilidades estadísticas

Vamos a ilustrar lo aprendido con el desarrollo de un módulo interesante: una colección de funciones que permitan realizar estadísticas de series de números, concretamente, el cálculo de la media, de la varianza y de la desviación típica.

Nuestro módulo debería utilizarse desde programas como se ilustra en este ejemplo:

uso_estadisticas.py
```
from estadisticas import media, desviación_típica

notas = []
nota = float(input('Dame una nota (entre 0 y 10): '))
while nota >= 0 and nota <= 10:
    notas.append(nota)
    nota = float(input('Dame una nota (entre 0 y 10): '))

print('Media:', media(notas))
print('Desviación típica:', desviación_típica(notas))
```

La media de una serie de números $a_1, a_2, \ldots, a_n$ es

$$\bar{a} = \frac{1}{n}\sum_{i=1}^{n} a_i,$$

su varianza es

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(a_i - \bar{a})^2.$$

y su desviación típica es

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(a_i - \bar{a})^2}.$$

Empecemos por el cálculo de la media:

estadisticas.py
```
def media(lista):
    suma = 0
    for elemento in lista:
        suma += elemento
    return suma / len(lista)
```

La varianza utiliza el valor de la media y podemos obtenerlo llamando a *media*:

estadisticas.py
```
def varianza(lista):
    suma = 0
    for elemento in lista:
        suma += (elemento - media(lista)) ** 2
```

```
    return suma / len(lista)
```

Mmmm. Está bien, pero se efectúa una llamada a *media* por cada iteración del bucle y hay tantas como elementos tiene la lista. Esa es una fuente de ineficiencia. Mejor calcular la media una sola vez y guardarla en una variable local:

```
estadisticas.py
def varianza(lista):
    suma = 0
    m = media(lista)
    for elemento in lista:
        suma += (elemento - m) ** 2
    return suma / len(lista)
```

Finalmente, la desviación típica no es más que la raíz cuadrada de la varianza, así que:

```
estadisticas.py
def desviación_típica(lista):
    return sqrt(varianza(lista))
```

---

384 ¿Funcionan bien las funciones que hemos definido cuando suministramos listas vacías? Corrige las funciones para que traten correctamente este caso particular.

385 Enriquece el módulo *estadisticas* añadiendo una función que calcule el coeficiente de variación (definido como $\sigma/\bar{a}$) y el recorrido de la lista (que es la diferencia entre el mayor y el menor elemento de la lista).

386 Suponiendo que nos suministran una lista de enteros, diseña una función que calcule su moda. La moda es el elemento más repetido en una serie de valores.

6.10.4. Un módulo para cálculo matricial

En el tema anterior estudiamos cómo operar con matrices. Vamos a «empaquetar» ahora algunas funciones útiles para manejar matrices.

Empezaremos por una función que crea una matriz nula dados su número de filas y columnas:

```
matrices.py
def matriz_nula(filas, columnas):
    M = []
    for i in range(filas):
        M.append( [0] * columnas )
    return M
```

Para crear una matriz A de dimensión 3×4 invocaremos así a la función:

```
A = matriz_nula(3, 4)
```

Ahora podemos escribir una función que lee de teclado los componentes de una matriz:

```
matrices.py
def lee_matriz(filas, columnas):
    M = matriz_nula(filas, columnas)
    for i in range(filas):
        for j in range(columnas):
            M[i][j] = float(input('Dime el componente ({0},{1}): '.format(i, j)))
    return M
```

Vamos ahora a por una función que sume dos matrices. Dos matrices *A* y *B* se pueden sumar si presentan la misma dimensión, es decir, el mismo número de filas y el mismo número de columnas. Nuestra función debería empezar comprobando este extremo. ¿Cómo podemos conocer la dimensión de una matriz *M*? El número de filas está claro: `len(M)`. ¿Y el número de columnas? Fácil, es el número de elementos de la primera fila (de cualquier fila, de hecho): `len(M[0])`. Expresar el número de filas y columnas como `len(M)` y `len(M[0])` no ayudará a hacer legible nuestra función de suma de matrices. Antes de empezar a escribirla, definamos una función que devuelva la dimensión de una matriz:

```
matrices.py
def dimensión(M):
    return [len(M), len(M[0])]
```

Para averiguar el número de filas y columnas de una matriz *A* bastará con hacer:

```
[filas, columnas] = dimensión(A)
```

▶ 387 Diseña una función llamada *es_cuadrada* que devuelva **True** si la matriz es cuadrada (tiene igual número de filas que columnas) y **False** en caso contrario. Sírvete de la función *dimensión* para averiguar la dimensión de la matriz.

Ahora, nuestra función de suma de matrices empezará comprobando que las matrices que se le suministran son «compatibles». Si no lo son, devolveremos **None** (ausencia de valor):

```
matrices.py
def suma(A, B):
    if dimensión(A) != dimensión(B):
        return None
    else:
        ...
```

Utilizaremos ahora la función *matriz_nula* para inicializar a cero la matriz resultante de la suma y efectuamos el cálculo (si tienes dudas acerca del procedimiento, consulta el tema anterior):

```
matrices.py
def suma(A, B):
    if dimensión(A) != dimensión(B):
        return None
    else:
        [m, n] = dimensión(A)
        C = crea_matriz_nula(m, n)
        for i in range(m):
            for j in range(n):
                C[i][j] = A[i][j] + B[i][j]
        return C
```

▶ 388 Enriquece el módulo `matrices.py` con una función que devuelva el producto de dos matrices. Si las matrices no son «multiplicables», la función devolverá **None**.

Capítulo 7

Tipos estructurados: clases y diccionarios

— No tendría un sabor muy bueno, me temo...

— Solo no —le interrumpió con cierta impaciencia el Caballero— pero no puedes imaginarte qué diferencia si lo mezclas con otras cosas...

Alicia en el país de las maravillas, Lewis Carroll

El conjunto de tipos de datos Python que hemos estudiado se divide en tipos *escalares* (enteros y flotantes) y tipos *secuenciales* (cadenas y listas). En este capítulo aprenderemos a definir y utilizar tipos de datos definidos por nosotros mismos *componiendo* otros tipos de datos más sencillos. Los nuevos tipos de datos reciben el nombre de *clases*.

Por otra parte, los *diccionarios* permiten mantener correspondencias entre claves y valores, facilitando así la escritura de ciertos programas.

7.1. Tipos de datos «a medida»

7.1.1. Lo que sabemos hacer

Supón que en un programa utilizamos el nombre, el DNI y la edad de dos personas. Necesitaremos tres variables para almacenar los datos de cada persona, dos variables con valores de tipo cadena (el nombre y el DNI) y otra con un valor de tipo entero (la edad):

```
nombre = 'Juan Pérez'
dni = '12345678Z'
edad = 19

otro_nombre = 'Pedro López'
otro_dni = '23456789D'
otra_edad = 18
```

Los datos almacenados en *nombre*, *dni* y *edad* corresponden a la primera persona y los datos guardados en *otro_nombre*, *otro_dni* y *otra_edad* corresponden a la segunda persona, pero nada en el programa permite deducir eso con seguridad: cada dato está almacenado en una variable *diferente e independiente* de las demás. El programador debe saber en todo momento qué variables están relacionadas entre sí y en qué sentido para utilizarlas coherentemente.

Diseñemos un procedimiento que muestre por pantalla los datos de una persona y usémoslo:

```
def imprimir_persona_en_pantalla(nombre, dni, edad):
    print('Nombre:', nombre)
    print('DNI:   ', dni)
    print('Edad:  ', edad)

```

```
imprimir_persona_en_pantalla(nombre, dni, edad)
imprimir_persona_en_pantalla(otro_nombre, otro_dni, otra_edad)
```

Al ejecutar ese fragmento de programa, por pantalla aparecerá:

```
Nombre: Juan Pérez
DNI:    12345678Z
Edad:   19
Nombre: Pedro López
DNI:    23456789D
Edad:   18
```

Funciona, pero resulta un tanto incómodo pasar tres parámetros cada vez que usamos el procedimiento. Si más adelante enriquecemos los datos de una persona añadiendo su domicilio, por ejemplo, tendremos que redefinir el procedimiento *imprimir_persona_en_pantalla* y cambiar todas sus llamadas para incluir el nuevo dato. En un programa de tamaño moderadamente grande puede haber decenas o cientos de llamadas a la función.

Imaginemos ahora que nuestro programa gestiona la relación de personas que asiste a clase. No sabemos a priori de cuántas personas estamos hablando, así que hemos de gestionar una lista a la que iremos añadiendo cuantas personas sea necesario. Bueno, *una* lista no, sino *tres* listas «paralelas»: una para los nombres, una para los DNI y una para las edades. Entenderemos que los elementos de las tres listas que tienen el mismo índice contienen los tres datos que describen a una persona en nuestro programa. Este fragmento de código ilustra la idea:

```
nombre = ['Juan␣Pérez', 'Pedro␣López', 'Ana␣García']
dni = ['12345678Z', '23456789D', '13577532B']
edad = [19, 18, 18]

for i in range(len(nombre)):
    imprimir_persona_en_pantalla(nombre[i], dni[i], edad[i])
```

El bucle recorre con *i* los índices 0, 1 y 2 y, para cada uno, muestra los tres datos asociados a una de las personas. Por ejemplo, cuando *i* vale 1 se muestran los datos de Pedro López, que están almacenados en *nombre*[1], *dni*[1] y *edad*[1]. Hemos ganado en comodidad (ya no hay que inventar un nombre de variable para cada dato de cada persona), pero hemos de estar atentos y mantener la coherencia entre las tres listas. Si, por ejemplo, queremos borrar los datos de Pedro López, tendremos que ejecutar tres operaciones de borrado (**del**):

```
del nombre[1]
del dni[1]
del edad[1]
```

Y si deseamos ordenar alfabéticamente la relación de personas por su nombre deberemos ser cuidadosos: cada intercambio de elementos de la lista *nombre*, supondrá el intercambio de los elementos correspondientes en las otras dos listas.

En resumen, es posible desarrollar programas que gestionan «personas» con esta metodología, pero resulta incómodo.

7.1.2. ... pero sabemos hacerlo mejor

Hay una alternativa a trabajar con grupos de tres variables independientes por persona: definir una «persona» como una lista con tres elementos. En cada elemento de la lista almacenaremos uno de sus valores, siempre en el mismo orden:

```
juan = ['Juan␣Pérez', '12345678Z', 19]
pedro = ['Pedro␣López,' '23456789D', 18]
```

Trabajar así permite que los datos de cada persona estén agrupados, sí, pero también hace algo incómodo su uso. Deberemos recordar que el índice 0 accede al nombre, el índice 1 al DNI y el índice 2 a la edad. Por ejemplo, para acceder a la edad de Juan Pérez hemos de escribir

juan[2]. Es probable que cometamos algún error difícil de detectar si utilizamos los índices erróneamente. Podríamos facilitar el trabajo almacenando los índices en unas variables cuyos identificadores permitan recordar sus respectivos «significados»:

```
nombre = 0
dni = 1
edad = 2

juan = ['Juan␣Pérez', '12345678Z', 19]
pedro = ['Pedro␣López,' '23456789D', 18]
```

Ahora, la edad de Juan Pérez se puede obtener escribiendo *juan*[*edad*], que es bastante más elegante que *juan*[2]. La función que muestra por pantalla todos los datos de una persona tendría este aspecto:

```
def imprimir_persona_en_pantalla(persona):
    print('Nombre:', persona[nombre])
    print('DNI:␣', persona[dni])
    print('Edad:␣', persona[edad])
```

Este procedimiento solo tiene un parámetro, así que, si añadimos nuevos datos a una persona, solo modificaremos el cuerpo del procedimiento, pero no todas y cada una de sus llamadas. Hemos mejorado, pues, con respecto a la solución desarrollada en el apartado anterior.

Siguiendo esta filosofía, también es posible tener listas de personas, que no serán más que listas de listas:

```
juan = ['Juan␣Pérez', '12345678Z', 19]
pedro = ['Pedro␣López,' '23456789D', 18]

personas = [juan, pedro]
```

Continuará

Seguro que a estas alturas ya te has encontrado con numerosas ocasiones en las que no te cabe una línea de programa Python en el ancho normal de la pantalla. No te preocupes: puedes partir una línea Python en varias para aumentar la legibilidad, aunque deberás indicarlo explícitamente. Una línea que finaliza con una barra invertida continúa en la siguiente:

```
a = 2 + \
    2
```

¡Ojo! La línea debe *acabar* en barra invertida, es decir, el carácter que sigue a la barra invertida \ debe ser el salto de línea (que es invisible). Si a la barra invertida le sigue un espacio en blanco, Python señalará un error.

O, directamente:

```
personas = [ ['Juan␣Pérez', '12345678Z', 19], \
             ['Pedro␣López,' '23456789D', 18] ]
```

(Si te sorprende la barra invertida al final de la primera línea, lee el cuadro «Continuará»).

El nombre de Pedro López, por ejemplo, está accesible en *personas*[1][*nombre*]. Si deseamos mostrar el contenido completo de la lista podemos hacer:

```
for persona in personas:
    imprimir_persona_en_pantalla(persona)
```

Solucionado. Bueno, no del todo. Si trabajamos con esta metodología nos aguardan nuevos problemas. Imagina que nuestro programa gestiona información sobre personas y coches y que de los coches necesitamos los siguientes datos: marca, modelo, matrícula y edad.

```
# Índices para personas
nombre = 0
dni = 1
edad = 2

juan  = ['Juan␣Pérez', '12345678Z', 19]
pedro = ['Pedro␣López,' '23456789D', 18]

# Índices para coches
modelo = 0
marca = 1
matrícula = 2
edad = 3        # Mal: edad ya estaba definida antes y valía 2

mi_coche = ['Biscúter', 'GTI', 'CS␣0000␣A', 42]
```

La variable *edad* ya estaba «ocupada» por «personas» y valía 2, así que asignarle ahora un valor distinto provocará errores cuando accedamos a la edad de una persona. ¿Cómo hacemos ahora para que la edad de un coche no se confunda con la edad de una persona? Tendremos que optar por:

- definir una nueva variable con otro nombre, por ejemplo, *edad_coche*, en la que almacenamos el índice correspondiente (el valor 3),
- o almacenar la edad del coche en la posición de índice 2 de la lista de datos del coche (es decir, poner la matrícula en la última posición de la lista y la edad en la penúltima).

Cualquiera de las dos posibilidades es «antinatural», pues estamos siendo obligados a tomar decisiones acerca de cómo representar una información (coche) motivadas por la existencia de otro tipo de información (persona) que nada tiene que ver con la primera. No es una buena solución.

7.1.3. Lo que haremos: usar tipos de datos «a medida»

Lo ideal sería que Python proporcionara un tipo de datos básico «persona» del mismo modo que proporciona datos enteros, flotantes, listas, etc. Una variable cuyo contenido fuera de tipo «persona» albergaría en un solo paquete toda la información propia de una persona: su nombre, su dni y su edad. Pero, claro, pronto pediremos que Python disponga de un tipo de datos «coche» (con la marca, el modelo, la matrícula y la edad, por ejemplo), de un tipo de datos «empleado» (con el nombre, nómina, categoría profesional y dirección de una persona), de un tipo de datos «profesor» (con su nombre, DNI, asignaturas impartidas y horarios de tutorías), etc., es decir, pediremos que Python incorpore un tipo de datos para cada conjunto de datos hipotéticamente necesario en nuestros programas.

Python no puede anticiparse a cualquier necesidad de cualquier programador proporcionando infinitos tipos de datos, pero sí nos permite *definir nuevos tipos de datos* combinando tipos de datos existentes.

Podemos definir un tipo de datos nuevo, digamos *Persona*, que agrupe en un solo paquete los datos básicos que lo forman: el nombre (una cadena), el DNI (otra cadena) y la edad (un entero). Fíjate en que el nuevo tipo de datos es una composición de tipos de datos existentes. Los nuevos tipos de datos recibirán el nombre genérico de *clases*.

Definir nuevos tipos de datos nos obligará a aprender nuevas construcciones sintácticas del lenguaje Python, pero antes será mejor que veamos con algunos ejemplos cómo se usarán los nuevos tipos. Supongamos que hemos definido la clase *Persona*. «Crearemos» nuevas personas así:

```
juan = Persona('Juan Pérez', '12345678Z', 19)
pedro = Persona('Pedro López', '23456789D', 18)
```

Si necesitamos acceder al nombre, DNI o edad de Juan Pérez, podremos hacerlo así:

```
>>> print(juan.nombre)
Juan Pérez
>>> print(juan.dni)
12345678Z
>>> print(juan.edad)
19
```

Fíjate: la variable *juan* contiene un dato de tipo *Persona* que, en realidad, son tres datos (el nombre, el DNI y la edad). Cada uno de dichos datos recibe el nombre de *atributo* o *campo*. Puedes consultar el valor de cada campo por su nombre separándolo del identificador de la variable con un punto.

El nuevo tipo de datos «sabrá» mostrarse en pantalla con la función *print*:

```
>>> print(juan)
Nombre: Juan Pérez
DNI: 12345678Z
Edad: 19
```

¡Mucho mejor que definir una función *imprime_persona_en_pantalla*!

Y aún más. Por ejemplo, imagina que necesitamos consultar con cierta frecuencia las iniciales de una persona. Podremos definir una función especial que efectúe el cálculo correspondiente a partir del nombre de la persona:

```
>>> print(juan.iniciales())
J. P.
```

Observa que *iniciales* se usa casi como si fuera el nombre de un campo de *juan*, pues se separa del identificador de la variable con un punto, pero se diferencia en el uso de paréntesis al final del identificador. Los paréntesis indican que *iniciales* es una función especial, un *método* que forma parte del tipo *Persona* y que lo estamos llamando (del mismo modo que se llama a una función). De todos modos, el uso de *iniciales* no debería resultarte demasiado extraño pues ya hemos usado métodos antes: al añadir un *dato* a una *lista* escribíamos *lista*.*append*(*dato*), es decir, llamábamos sobre *lista* al método *append* con el dato que deseábamos añadir.

Para acabar esta exposición introductoria abordaremos el caso de la relación de estudiantes que asisten a clase y que antes resolvimos usando tres listas paralelas. Al disponer ahora de un tipo de datos *Persona* solo es necesario disponer de una lista:

```
alumnos = [Persona('Antonio Pérez', '98761234Q', 20), \
           Persona('Juan Pérez', '12345678Z', 19), \
           Persona('Pedro López', '23456789D', 18)]
```

El siguiente fragmento de código mostrará los datos de todas las personas de la lista:

```
for i in range(len(alumnos)):
    print(alumnos[i])
```

O, alternativamente:

```
for alumno in alumnos:
    print(alumno)
```

Elegante, ¿no?

7.2. Definición de clases en Python

Vamos a definir un nuevo tipo de datos Python: la clase *Persona*. Presta atención a la sintaxis: es un poco enrevesada a primera vista.

persona.py
```python
1 class Persona:
2     def __init__(self, nombre, dni, edad):
3         self.nombre = nombre
4         self.dni = dni
5         self.edad = edad
```

Analicémosla por partes. La primera línea empieza con la palabra reservada **class** y con ella indicamos que comienza la definición de un nuevo tipo de datos, de una nueva clase. A continuación aparece el identificador del nuevo tipo (*Persona*) y dos puntos. Las líneas 2 y siguientes empiezan más a la derecha y definen una función (empieza por **def**) llamada *__init__*. Las funciones que definimos *dentro* de una clase se denominan *métodos*. El método *__init__* tiene cuatro parámetros: el primero se llama *self*. *Todos los métodos tendrán como primer parámetro uno llamado self*. Le siguen tantos parámetros como campos tiene una variable del tipo *Persona*.[1] El cuerpo del método *__init__* (líneas 3 a 5) consiste en una serie de asignaciones de la forma:

```
self.atributo = parámetro
```

¿Qué es *self*? En inglés, «*self*» significa «uno mismo». Al asignar a *self*.*nombre* el valor del parámetro *nombre* estamos diciendo algo así como «el *nombre* de uno mismo es el valor del parámetro *nombre*». Es la forma de almacenar información en «uno mismo».

Ya puedes almacenar personas en variables. Cada vez que quieras crear una nueva persona, deberás hacerlo así:

```python
toni = Persona('Antonio Pérez', '987612340Q', 20)
```

Cada vez que creas o «construyes» una nueva persona, Python llama automáticamente al método *__init__*. El método *__init__* es el denominado *constructor* de la clase *Persona*. Python interpreta esa sentencia como:

```python
toni.nombre = 'Antonio Pérez'
toni.dni = '98761234Q'
toni.edad = 20
```

pues *self* equivale a *toni* en el ejemplo.

Cada persona creada es una *instancia* u *objeto* de la clase *Persona*. Puedes utilizar objetos de la clase *Persona* del mismo modo que utilizabas valores de otros tipos. Por ejemplo, puedes crear «personas» y almacenarlas en variables y/o en listas, a voluntad:

```python
toni = Persona('Antonio Pérez', '98761234Q', 20)
juan = Persona('Juan Pérez', '12345678Z', 19)
pedro = Persona('Pedro López', '23456789D', 18)
alumnos = [toni, juan, pedro]
```

Si deseas acceder a la edad de *toni*, podrás utilizar la notación introducida en el apartado anterior:

```
>>> print(toni.edad)
20
>>> print(alumnos[0].edad)
20
```

Puedes acceder a los elementos de la lista *alumnos* como siempre. Este fragmento de programa, por ejemplo, muestra el dni de los integrantes de la lista:

[1] Más adelante veremos que no es necesario que haya tantos parámetros como atributos.

POO

Este apartado constituye una introducción a uno de los aspectos básicos de la Programación Orientada a Objetos (POO). La POO es un paradigma (una forma) de programar muy extendida en las últimas décadas. Los otros dos pilares de la POO son la *herencia* y el *polimorfismo*, aunque no los trataremos en este curso.

Python es un lenguaje orientado a objetos, es decir, da soporte a las características propias de la POO. Entre los lenguajes orientados a objetos de uso más extendido se cuentan C++ y Java.

```
for alumno in alumnos:
    print(alumno.dni)
```

Alternativamente puedes recorrer la lista así:

```
for i in range(len(alumnos)):
    print(alumnos[i].dni)
```

Observa que en este caso hemos aplicado primero el operador de indexación a la lista y luego hemos accedido al campo *dni*. El contenido de *alumnos*[*i*] es una *Persona*, así que podemos añadir .*dni* para acceder a su campo *dni*.

▶ 389 Diseña un programa que pida por teclado los datos de varias personas y los añada a una lista inicialmente vacía. Cada vez que se lean los datos de una persona el programa preguntará si se desea continuar introduciendo nuevas personas. Cuando el usuario responda que no, el programa se detendrá.

▶ 390 Modifica el programa del ejercicio anterior para que, a continuación, muestre el nombre de la persona de más edad. Si dos o más personas tienen la mayor edad, el programa mostrará el nombre de todas ellas.

Dotemos a los objetos de la clase *Persona* de cierta «inteligencia»: hagamos que sepan devolvernos las iniciales de su nombre. Definiremos un método *iniciales* que devuelva una cadena:

persona.py
```
class Persona:
    def __init__(self, nombre, dni, edad):
        self.nombre = nombre
        self.dni = dni
        self.edad = edad

    def iniciales(self):
        cadena = ''
        for carácter in self.nombre:
            if carácter >= 'A' and carácter <= 'Z':
                cadena = cadena + carácter + '. '
        return cadena
```

Ahora nos detendremos a explicar paso a paso cómo hemos definido el nuevo método, pero antes, veamos si funciona:

```
>>> print(juan.iniciales())
J. P.
```

¡Perfecto! Ya te habíamos dicho que todos los métodos tienen un primer parámetro llamado *self*, e *iniciales* no es una excepción. Cuando efectuamos una llamada a un método siempre lo haremos «sobre» una variable del tipo *Persona* y, en ese caso, *self* se interpreta como dicha variable. Cuando hemos ejecutado *juan*.*iniciales*(), el parámetro *self* se ha interpretado como *juan*, así que el acceso a *self*.*nombre* es, en ese caso, un acceso a *juan*.*nombre*, es decir, un acceso al atributo *nombre* de «uno mismo».

▶ 391 Diseña un método que permita determinar si una persona es menor de edad devolviendo cierto, si la edad es menor que 18, o falso, en caso contrario.

▶ 392 Diseña un método *nombre_de_pila* que devuelva el nombre de pila de una *Persona*. Supondremos que el nombre de pila es la primera palabra del atributo *nombre* (es decir, que no hay nombres compuestos).

¿Qué pasa si mostramos con *print* una variable de tipo *Persona*?

```
>>> print(juan)↵
<__console__.Persona object at 0x3cc2a50>
```

Mal. No sale lo que esperamos. Definamos un nuevo método que permitirá imprimir objetos de la clase *Persona*:

```python
# persona.py
class Persona:
    def __init__(self, nombre, dni, edad):
        self.nombre = nombre
        self.dni = dni
        self.edad = edad

    def iniciales(self):
        cadena = ''
        for carácter in self.nombre:
            if carácter >= 'A' and carácter <= 'Z':
                cadena = cadena + carácter + '. '
        return cadena

    def __str__(self):
        cadena = 'Nombre: {0}\n'.format(self.nombre)
        cadena = cadena + 'DNI: {0}\n'.format(self.dni)
        cadena = cadena + 'Edad: {0}\n'.format(self.edad)
        return cadena
```

Mmmm. Un método llamado *__str__*. ¡Qué nombre tan extraño! Bueno, ¿funciona ahora?

```
>>> print(juan)↵
Nombre: Juan Pérez
DNI: 12345678Z
Edad: 19
```

¡Ahora sí! Ciertos métodos tienen nombres especiales, como *__init__* y *__str__*. Python espera que un método con un nombre especial haga algo concreto. Por ejemplo, un método llamado *__init__* debe construir un objeto de la clase *Persona* y un método llamado *__str__* debe devolver una *cadena* con lo que queremos que se muestre por pantalla (el *str* del nombre del método es una abreviatura de «string», es decir, «cadena» en inglés). En nuestro caso, la cadena que hemos formado contiene todos los datos de una *Persona*. Lo realmente curioso acerca de los métodos especiales es que no tienes por qué llamarlos directamente: Python los llama automáticamente en ciertos casos. Por ejemplo, cuando haces *print* de un objeto de la clase *Persona*, Python le «pregunta» al objeto si tiene definido el método *__str__* y, si es así, muestra el resultado de ejecutar dicho método sobre el objeto. Como el resultado de ejecutar *__str__* es una cadena, Python muestra por pantalla esa cadena.

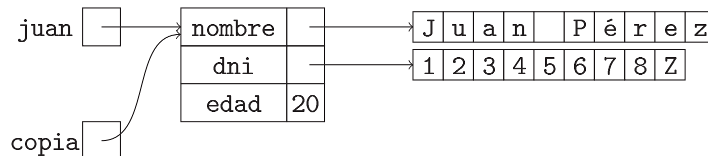
▶ 393 Modifica el programa del ejercicio anterior enriqueciendo el tipo de datos *Persona* con un nuevo campo: el sexo, que codificaremos con una letra (`'M'` para mujer y `'V'` para varón). Añade a tu programa un método *__str__* que también imprima en pantalla cuál es el sexo de la persona.

7.2.1. Referencias y objetos

Hemos de tratar ahora un problema que ya nos es conocido. Fíjate bien:

```
>>> juan = Persona('Juan␣Pérez', '12345678Z', 19)↵
>>> copia = juan↵
>>> copia.edad = 20↵
>>> print(copia.edad)↵
20
>>> print(juan.edad)↵
20
```

¡Los cambios a *copia* afectan a *juan*! Estamos ante el mismo problema que apareció al trabajar con listas: la asignación de un objeto a otro *solo copia la referencia*, y no el contenido.



No solo la asignación se ve afectada por este hecho: también el paso de parámetros se efectúa transmitiendo a la función una referencia al objeto, así que *los cambios realizados a un objeto dentro de una función son «visibles» fuera, en el objeto pasado como argumento.*

Al trabajar con listas pudimos solucionar el problema obteniendo una copia de la lista a asignar con el operador de corte o la concatenación. Pero el operador de corte no tiene significado alguno para nuestros objetos. ¿Cómo solucionar el problema? Lo normal es que definamos un método capaz de generar una copia de nuestro objeto y lo usemos cuando sea menester:

persona.py
```
class Persona:

    ...

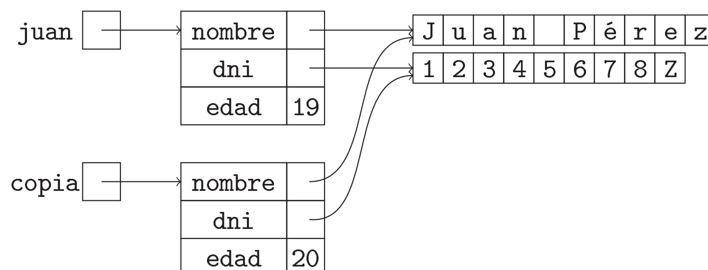
    def copia(self):
        nuevo = Persona(self.nombre, self.dni, self.edad)
        return nuevo
```

Observa que hemos creado y devuelto una nueva *Persona* cuyos atributos tienen los mismos valores que tiene *self*, es decir, la *Persona* sobre la que invocamos el método.

Repitamos la prueba anterior:

```
>>> juan = Persona('Juan␣Pérez', '12345678Z', 19)↵
>>> copia = juan.copia()↵
>>> copia.edad = 20↵
>>> print(copia.edad)↵
20
>>> print(juan.edad)↵
19
```

¡Ahora sí! ¿Cómo ha quedado la memoria en este caso? Observa detenidamente la siguiente figura:



La verdad es que *juan* y *copia* ahora apuntan a objetos de la clase *Persona* diferentes, pero no completamente independientes: ¡siguen compartiendo la memoria de las cadenas! ¿Por qué? Recuerda que la asignación de una cadena a otra se traduce en la copia de la referencia, no del contenido, y cuando se ejecuta *copia*.*nombre* = *juan*.*nombre*[2] el valor de *copia*.*nombre* es una referencia a la memoria apuntada por *juan*.*nombre*.

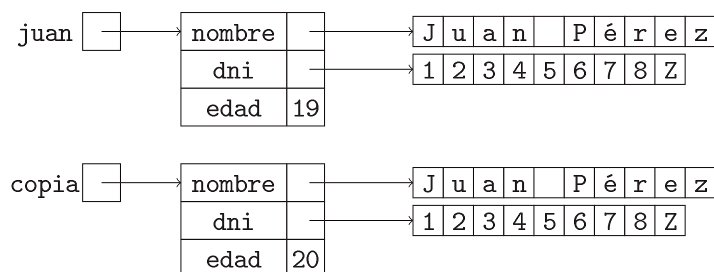
Como las cadenas son inmutables en Python, no hay problema alguno en que *juan* y *copia* compartan la memoria de sus campos *nombre* y *dni*. Aun así, si quisiéramos que ambos tuvieran su propia zona de memoria para estos datos, deberíamos modificar el método de copia de la clase *Persona*:

persona.py
```
class Persona:

    ...

    def copia(self):
        nuevo = Persona(self.nombre[:], self.dni[:], self.edad)
        return nuevo
```

Tras ejecutar las sentencias del ejemplo con el nuevo método *copia* tenemos:



La gestión de la memoria es un asunto delicado y la mayor parte de los errores graves de programación están causados por un inapropiado manejo de la memoria. Python simplifica mucho dicha gestión, pero un programador competente debe saber qué ocurre exactamente en memoria cada vez que se maneja una cadena, lista u objeto.

▶ 394 ¿Qué mostrará por pantalla la ejecución del siguiente programa?

```
class Persona:
    ...

def nada_útil(persona1, persona2):
    persona1.edad += 1
    persona3 = persona2
    persona4 = persona2.copia()
    persona3.edad -= 1
    persona4.edad -= 2
    return persona4

juan = Persona('Juan Pérez', '12345679Z', 19)
pedro = Persona('Pedro López', '23456789D', 18)
otro = nada_útil(juan, pedro)
print(juan)
print(pedro)
print(otro)
```

[2]En realidad, esta sentencia de asignación aparece en el programa expresada como *self*.*nombre* = *nombre* en el método *__init__* que se ejecuta al construir *copia*.

Haz un diagrama que muestre el estado de la memoria en los siguientes instantes:

a) justo antes de ejecutar la línea 14,

b) justo antes de ejecutar la línea 10 en la invocación de *nada_útil* desde la línea 14,

c) al finalizar la ejecución del programa.

7.2.2. Un ejemplo: gestión de calificaciones de estudiantes

Desarrollemos un ejemplo completo. Vamos a diseñar un programa que gestiona la lista de estudiantes de una asignatura y sus calificaciones. De cada estudiante guardaremos su nombre, su grupo de teoría (que será la letra A, B o C), la nota obtenida en el examen y si ha entregado o no las prácticas. Tener aprobada la asignatura implica haber entregado las prácticas y haber obtenido en el examen una nota igual o superior a 5.

Deseamos hacer un programa que permita añadir estudiantes a la lista, mostrar la calificación de cada uno de ellos y efectuar algunas estadísticas sobre las notas, como obtener la nota media o el porcentaje de estudiantes que ha entregado las prácticas.

Definamos primero el tipo de datos *Estudiante*. Cada estudiante tiene cuatro campos (*nombre*, *grupo*, *nota* y *práctica*):

```
notas.py
class Estudiante:
    def __init__(self, nombre, grupo, nota, práctica):
        self.nombre = nombre
        self.grupo = grupo
        self.nota = nota
        self.práctica = práctica
```

Los campos *nombre* y *grupo* serán cadenas, el campo *nota* será un flotante con el valor numérico de la evaluación del examen y el valor del campo *práctica* será True si la entregó y False en caso contrario.

Sería interesante definir una función que leyera por teclado los datos de un estudiante y nos devolviera un nuevo objeto *Estudiante* con sus campos cumplimentados.

```
notas.py
def lee_estudiante():
    nombre = input('Nombre:␣')
    grupo = input('Grupo␣(A,␣B␣o␣C):␣')
    nota = float(input('Nota␣de␣examen:␣'))
    entregada = input('Práctica␣entregada␣(s/n):␣')
    práctica = entregada == 's'
    return Estudiante(nombre, grupo, nota, práctica)
```

¡Ojo! *lee_estudiante* no es un método, sino una función, y como tal se define *fuera* de la clase *Estudiante*. La función lee de teclado el valor de cada campo y construye un nuevo *Estudiante*, que es el valor que devuelve. Podemos pedir al usuario de nuestro programa que introduzca los datos de un estudiante así:

```
nuevo_estudiante = lee_estudiante()
```

El contenido de *nuevo_estudiante* es un objeto de la clase *Estudiante*.

Diseñemos ahora una función que, dada una lista de estudiantes (posiblemente vacía), pida los datos de un estudiante y añada el nuevo estudiante a la lista:

```
notas.py
def lee_y_añade_estudiante(lista):
    estudiante = lee_estudiante()
```

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

```
        lista.append(estudiante)
```

Definamos ahora el método __str__ para poder imprimir en pantalla un estudiante:

notas.py
```
class Estudiante:
    ...

    def __str__(self):
        cadena = 'Nombre:␣{0}\n'.format(self.nombre)
        cadena = cadena + 'Grupo:␣{0}\n'.format(self.grupo)
        cadena = cadena + 'Nota␣examen:␣{0:3.1f}\n'.format(self.nota)
        if self.práctica:
            cadena = cadena + 'Práctica␣entregada'
        else:
            cadena = cadena + 'Práctica␣no␣entregada'
        return cadena
```

▶ 395 Diseña un procedimiento que, dada una lista de estudiantes, muestre por pantalla los datos de todos ellos.

▶ 396 Diseña un procedimiento que, dada una lista de estudiantes y un grupo (la letra A, B o C), muestre por pantalla un listado completo de dicho grupo.

Ahora nos gustaría conocer la calificación de un estudiante: Matrícula de Honor, Notable, Aprobado o Suspenso. No existe un campo *calificación* en los objetos de la clase *Estudiante*, así que deberemos implementar un método que efectúe los cálculos pertinentes a partir del valor de *práctica* y del valor de *nota*:

notas.py
```
class Estudiante:
    ...

    def calificación(self):
        if not self.práctica:
            return 'Suspenso'
        else:
            if self.nota < 5:
                return 'Suspenso'
            elif self.nota < 7:
                return 'Aprobado'
            elif self.nota < 8.5:
                return 'Notable'
            elif self.nota < 10:
                return 'Sobresaliente'
            else:
                return 'Matrícula␣de␣Honor'
```

Probemos si funciona:

```
>>> pepe = Estudiante('Pepe␣García', 'A', 7.7, True)↵
>>> print(pepe.calificación())↵
Notable
```

▶ 397 Define un método *está_aprobado* que devuelva True si el alumno ha aprobado la asignatura y False en caso contrario.

Podemos escribir ahora una función que muestre el nombre y la calificación de todos los estudiantes:

```
notas.py
def acta(lista):
    for estudiante in lista:
        print(estudiante.nombre, estudiante.calificación())
```

Si queremos obtener algunas estadísticas, como la nota media o el porcentaje de estudiantes que ha entregado las prácticas, definiremos y usaremos nuevas funciones:

```
notas.py
def nota_media(lista):
    suma = 0
    cantidad = 0
    for estudiante in lista:
        if estudiante.práctica:
            suma += estudiante.nota
            cantidad += 1
    if cantidad != 0:
        return suma / cantidad
    else:
        return 0.0

def porcentaje_de_prácticas_entregadas(lista):
    if len(lista) != 0:
        cantidad = 0
        for estudiante in lista:
            if estudiante.práctica:
                cantidad += 1
        return cantidad / len(lista) * 100
    else:
        return 0.0
```

▶ 398 Diseña una función que devuelva el porcentaje de aprobados sobre el total de estudiantes (y no sobre el total de estudiantes que han entregado la práctica).

▶ 399 Diseña una función que reciba una lista de estudiantes y el código de un grupo (la letra A, B o C) y devuelva la nota media en dicho grupo.

Y esta otra función, por ejemplo, devuelve una lista con los estudiantes que obtuvieron la nota más alta:

```
notas.py
def mejores_estudiantes(lista):
    nota_más_alta = 0
    mejores = []
    for estudiante in lista:
        if estudiante.práctica:
            if estudiante.nota > nota_más_alta:
                mejores = [estudiante]
                nota_más_alta = estudiante.nota
            elif estudiante.nota == nota_más_alta:
                mejores.append(estudiante)
    return mejores
```

Fíjate en que *mejores_estudiantes* devuelve una lista cuyos componentes son objetos de la clase *Estudiante*. Si deseas listar por pantalla los nombres de los mejores estudiantes, puedes hacer lo siguiente:

```
los_mejores = mejores_estudiantes(lista)
for estudiante in los_mejores:
    print(estudiante.nombre)
```

o, directamente:

```
for estudiante in mejores_estudiantes(lista):
    print(estudiante.nombre)
```

▶ 400 Deseamos realizar un programa que nos ayude a gestionar nuestra colección de ficheros MP3. Cada fichero MP3 contiene una canción y deseamos almacenar en nuestra base de datos la siguiente información de cada canción:

- título,
- intérprete,
- duración en segundos,
- estilo musical.

Empieza definiendo la clase *MP3* y su método *__init__*. Cuando lo tengas, define dos nuevos métodos:

- *resumen*: devuelve una cadena con solo el título y el intérprete (en una sola línea).
- *__str__*: devuelve una cadena con todos los datos del fichero MP3 de modo que cada campo ocupe una línea.

A continuación, diseña cuantos métodos y funciones consideres pertinentes para implementar las siguientes acciones:

a) añadir una nueva canción a la base de datos (que será una lista de objetos de la clase *MP3*),

b) listar todas las canciones de un intérprete determinado (en formato resumido),

c) listar todas las canciones de un estilo determinado (en formato resumido),

d) listar todas las canciones de la base de datos (en formato completo),

e) eliminar una canción de la base de datos dado el título y el intérprete.

(Nota: Si quieres que el programa sea realmente útil, sería interesante que pudieras salvar la lista de canciones a disco duro; de lo contrario perderás todos los datos cada vez que salgas del programa. En el próximo capítulo aprenderemos a guardar datos en disco y a recuperarlos, así que este programa solo te resultará realmente útil cuando hayas estudiado ese capítulo. De momento, si quieres usarlo ya, puedes utilizar el módulo *pickle* que describimos sucintamente en el capítulo anterior).

7.3. Algunas clases de uso común

Muchas aplicaciones utilizan ciertos tipos de datos estructurados. Un principio de diseño es la reutilización de código, es decir, no reescribir lo que ya hemos implementado cada vez que necesitemos usarlo. Nos vendrá bien disponer de módulos en los que hayamos implementado estas clases de datos. De ese modo, cada aplicación que necesite utilizar el tipo de datos en cuestión, solo tendrá que importar la clase correspondiente del módulo.

7.3.1. La clase fecha

Python no dispone de un tipo de datos fecha, y la verdad es que nos vendría bien en numerosas aplicaciones. Vamos a diseñar una clase *Fecha* y la implementaremos en un módulo *fecha* (es decir, en un fichero **fecha.py**).

Una fecha tiene tres valores: día, mes y año. Codificaremos cada uno de ellos con un número entero.

fecha.py
```
class Fecha:
    def __init__(self, día, mes, año):
        self.día = día
        self.mes = mes
        self.año = año
```

Mmmm. Seguro que nos viene bien un método que muestre por pantalla una fecha.

fecha.py
```
class Fecha:
    ...

    def __str__(self):
        return '{0}/{1}/{2}'.format(self.día, self.mes, self.año)
```

▶ 401 Define un método llamado *formato_largo* que devuelva la fecha en un formato más verboso. Por ejemplo, el 15/4/2002 aparecerá como `15 de abril de 2002`.

Definamos ahora un método que indica si un año es bisiesto o no. Recuerda que un año es bisiesto si es divisible por 4, excepto si es divisible por 100 pero no por 400:

fecha.py
```
class Fecha:
    ...

    def en_año_bisiesto(self):
        return self.año % 4 == 0 and (self.año % 100 != 0 or self.año % 400 == 0)
```

▶ 402 Diseña un método *válida* que devuelva cierto si la fecha es válida y falso en caso contrario. Para ello, debes comprobar que el mes esté comprendido entre 1 y 12 y que el día esté comprendido entre 1 y el número de días que corresponde al mes. Por ejemplo, la fecha 31/4/2000 no es válida, ya que abril tiene 30 días.

Ten especial cuidado con el mes de febrero: ¡tiene 28 o 29 días según el año! Usa, si te conviene, el método definido en el ejercicio anterior haciendo *self.en_año_bisiesto*().

Diseñemos ahora una función (no un método) que lee una fecha por teclado y nos la devuelve:

fecha.py
```
class Fecha:
    ...

def lee_fecha():
    día = 0
    while día < 1 or día > 31:
        día = int(input('Día:␣'))
    mes = 0
    while mes < 1 or mes > 12:
        mes = int(input('Mes:␣'))
    año = int(input('Año:␣'))
    return Fecha(día, mes, año)
```

▶ 403 Modifica la función *lee_fecha* para que solo acepte fechas válidas, es decir, fechas cuyo día sea válido para el mes leído. Puedes utilizar el método *válida* desarrollado en el ejercicio anterior.

Nos gustaría comparar dos fechas para saber si una es menor que otra. Podemos diseñar una función o un método. Implementemos ambas y así remarcaremos las diferencias entre función y método. Empecemos por la función:

```
fecha.py
def fecha_es_menor(fecha1, fecha2):
    if fecha1.año < fecha2.año:
        return True
    elif fecha1.año > fecha2.año:
        return False
    if fecha1.mes < fecha2.mes:
        return True
    elif fecha1.mes > fecha2.mes:
        return False
    return fecha1.día < fecha2.día
```

Si en un programa deseamos comparar dos fechas, *f1* y *f2*, lo haremos así:

```
...
if fecha_es_menor(f1, f2):
    ...
```

Vamos ahora a por el método:

```
fecha.py
class Fecha:
    ...

    def es_menor_que(self, la_otra_fecha):
        if self.año < la_otra_fecha.año:
            return True
        elif self.año > la_otra_fecha.año:
            return False
        if self.mes < la_otra_fecha.mes:
            return True
        elif self.mes > la_otra_fecha.mes:
            return False
        return self.día < la_otra_fecha.día
```

Observa que también tiene dos parámetros, pero el primero es *self*, es decir, «uno mismo», y el otro es la segunda fecha, «la otra». ¿Cómo se usa el método? Así:

```
...
if f1.es_menor_que(f2):
    ...
```

A gusto del consumidor.

▶ 404 Diseña un método que devuelva cierto si dos fechas son iguales y falso en caso contrario.

▶ 405 Diseña un método *añade_un_día* que añade un día a una fecha dada. La fecha 7/6/2001, por ejemplo, pasará a ser 8/6/2002 tras invocar al método *añade_un_día* sobre ella.

¿Cuántos días han pasado... dónde?

Trabajar con fechas tiene sus complicaciones. Una función que calcule el número de días transcurridos entre dos fechas cualesquiera no es trivial. Por ejemplo, la pregunta no se puede responder si no te dan otro dato: ¡el país! ¿Sorprendido? No te vendrá mal conocer algunos hechos sobre el calendario.

Para empezar, no existe el año cero, pues el cero se descubrió en Occidente bastante más tarde (en el siglo IX fue introducido por los árabes, que lo habían tomado previamente del sistema indio). El año anterior al 1 d. de C. (después de Cristo) es el 1 a. de C. (antes de Cristo). En consecuencia, el día siguiente al 31 de diciembre de 1 a. de C. es el 1 de enero de 1 d. de C.. (Esa es la razón de que el siglo XXI empezara el 1 de enero de 2001, y no de 2000, como erróneamente creyó mucha gente).

Julio César, en el año 46 a. C. difundió el llamado calendario juliano. Hizo que los años empezaran en 1 de januarius (el actual enero) y que los años tuvieran 365 días, con un año bisiesto cada 4 años, pues se estimaba que el año tenía 365,25 días. El día adicional se introducía tras el 23 de febrero, que entonces era el sexto día de marzo, con lo que aparecía un día "bis-sexto" (o sea, un segundo día sexto) y de ahí viene el nombre bisiesto de nuestros años de 366 días. Como la reforma se produjo en un instante en el que ya se había acumulado un gran error, Julio César decidió suprimir 80 días de golpe.

Pero la aproximación que del número de días de un año hace el calendario juliano no es exacta (un año dura en realidad 365,242198 días, 11 minutos menos de lo estimado) y comete un error de 7,5 días cada 1000 años. En 1582 el papa Gregorio XIII promovió la denominada reforma gregoriana con objeto de corregir este cálculo inexacto. Este papa suprimió los bisiestos seculares (los que corresponden a años divisibles por 100), excepto los que caen en años múltiplos de 400, que siguieron siendo bisiestos. Para cancelar el error acumulado por el calendario juliano, Gregorio XIII suprimió 10 días de 1582: el día siguiente al 4 de octubre de 1582 fue el 15 de octubre de 1582. Como la reforma fue propuesta por un papa católico, tardó en imponerse en países protestantes u ortodoxos. Inglaterra, por ejemplo, tardó 170 años en adoptar el calendario gregoriano. En 1752 ya se había producido un nuevo día de desfase entre el cómputo juliano y el gregoriano, así que se suprimieron 11 días del calendario: al 2 de septiembre de 1752 siguió en Inglaterra el 14 de septiembre del mismo año. Por otra parte, Rusia no adoptó el nuevo calendario hasta ¡1918!, así que la revolución de su octubre de 1917 tuvo lugar en nuestro noviembre de 1917. Y no fue Rusia el último país occidental en adoptar el calendario gregoriano: Rumanía aún tardó un año más.

Por cierto, el calendario gregoriano no es perfecto: cada 3000 años (aproximadamente) se desfasa en 1 día. ¡Menos mal que no nos tocará vivir la próxima reforma!

Atención al último día de cada mes, pues su siguiente día es el primero del mes siguiente. Similar atención requiere el último día del año. Debes tener en cuenta que el día que sigue al 28 de febrero es el 29 del mismo mes o el 1 de marzo dependiendo de si el año es bisiesto o no.

▶ 406 Diseña un método que calcule el número de días transcurridos entre la fecha sobre la que se invoca el método y otra que se proporciona como parámetro. He aquí un ejemplo de uso:

```
>>> ayer = Fecha(1, 1, 2002)↵
>>> hoy = Fecha(2, 1, 2002)↵
>>> print(hoy.días_transcurridos(ayer))↵
1
```

(No tengas en cuenta el salto de fechas producido como consecuencia de la reforma gregoriana del calendario. Si no sabes de qué estamos hablando, consulta el cuadro «¿Cuántos días han pasado... dónde?»).

▶ 407 Modifica el método anterior para que sí tenga en cuenta los 10 días perdidos en la reforma gregoriana... en España.

▶ 408 Diseña un método que devuelva el día de la semana (la cadena `'lunes'`, o `'martes'`, etc.) en que cae una fecha cualquiera. (Si sabes en que día cayó una fecha determinada, el número de días transcurridos entre esa y la nueva fecha módulo 7 te permite conocer el día de la semana).

7.3.2. Colas y pilas

En numerosas aplicaciones hemos de gestionar colas. Por ejemplo, en un programa de ayuda a la gestión de una consulta médica necesitamos manejar una cola de pacientes; o en la gestión de una lista de espera (una cola) de pasajeros en los vuelos con *overbooking*. Sería deseable disponer de un tipo de datos *Cola* que podamos utilizar en cualquier aplicación en la que haga falta. La cola podría comportarse como se muestra en este ejemplo (que, por simplificar, trabaja con números enteros):

```
>>> cola = Cola()↵
>>> cola.añade(5)↵
>>> cola.añade(8)↵
>>> cola.añade(4)↵
>>> print(cola)↵
5 8 4
>>> print(cola.primero())↵
5
>>> print(cola)↵
5 8 4
>>> cola.sacaPrimero()↵
>>> print(cola)↵
8 4
>>> cola.añade(9)↵
>>> print(cola)↵
8 4 9
>>> print(cola.tamaño())↵
3
>>> print(cola.esVacía())↵
False
>>> cola.sacaPrimero()↵
>>> cola.sacaPrimero()↵
>>> cola.sacaPrimero()↵
>>> print(cola)↵

>>> print(cola.esVacía())↵
True
>>> print(cola.primero())↵
None
```

Fíjate:

- La cola se construye sin ningún argumento (*cola* = *Cola*()): inicialmente la cola está vacía.
- El método *añade* permite añadir elementos al final de la cola.
- La función *print* muestra todos los elementos de la cola, empezando por el que entró en ella en primer lugar. Si la cola está vacía, no muestra nada.
- El método *primero* nos dice quién ocupa la primera posición de la cola. Si la cola está vacía, devuelve None.
- El método *sacaPrimero* elimina al primer elemento de la cola. Si la cola está vacía, no hace nada.
- El método *tamaño* nos dice cuántos elementos hay en la cola.
- El método *esVacía* nos dice si la cola está vacía o no, devolviendo cierto o falso.

Nuestro primer problema es decidir cómo guardamos la información propia de una cola y la respuesta es muy fácil: con una lista.

cola.py
```
class Cola:
    def __init__(self):
        self.cola = []
```

Hemos guardado la lista en el atributo *cola* y la hemos inicializado con la lista vacía. El constructor *__init__* no necesita parámetro alguno (salvo, *self*, naturalmente), pues una cola nueva siempre empieza estando vacía.

El método *añade* es sencillo de implementar: se limita a añadir a la lista un nuevo elemento por el final.

```
cola.py
class Cola:
    ...

    def añade(self, elemento):
        self.cola.append(elemento)
```

El método *primero* nos dice quién ocupa la primera posición de la cola, es decir, quién ocupa *self.cola*[0]:

```
cola.py
class Cola:
    ...

    def primero(self):
        if len(self.cola) == 0:
            return None
        else:
            return self.cola[0]
```

Y el método *sacaPrimero* sencillamente borra el primer elemento de la lista:

```
cola.py
class Cola:
    ...

    def sacaPrimero(self):
        if len(self.cola) > 0:
            del self.cola[0]
```

El resto de métodos no plantea dificultad alguna, así que los mostramos todos sin más dilación:

```
cola.py
class Cola:
    ...

    def __str__(self):
        cadena = ''
        for elemento in self.cola:
            cadena = cadena + str(elemento) + ' '
        return cadena

    def tamaño(self):
        return len(self.cola)

    def esVacía(self):
        return len(self.cola) == 0
```

▶ 409 Diseña un método *copia* que devuelva una nueva *Cola* cuyo contenido es el mismo de la *Cola* sobre la que se invoca el método. ¡Ojo con la memoria cuando saques una copia de *self.cola*!

▶ 410 Diseña un programa que gestiona una lista de espera de pacientes usando la clase *Cola* definida en este apartado. Cada paciente tiene un nombre y un número de la seguridad

social. El programa presentará un menú con las siguientes opciones: 1) añadir un paciente a la lista de espera; 2) atender al primer paciente de la lista; y 3) finalizar la ejecución del programa. Al seleccionar un paciente se solicitarán los datos del paciente y se añadirá este a la lista de espera (que es una cola). Al seleccionar la segunda opción, aparecerá en pantalla el nombre del paciente y se eliminará a este de la cola. La lista de espera respetará estrictamente el orden de llegada de los pacientes.

▶ 411 Modifica el programa anterior para que gestione dos colas: una para atención médica normal y otra para urgencias. Al añadir un paciente se preguntará si se trata de una urgencia o no. En el primer caso, se añadirá a una lista de espera de urgencias y en el segundo, a una lista de espera normal. Cada vez que se seleccione la segunda opción del menú aparecerá el nombre de un paciente en pantalla, pero tendrán preferencia aquellos que estén en la lista de espera de urgencias. Ningún paciente de la cola normal será atendido mientras haya uno solo en la cola de urgencias. Dentro de cada cola se respetará estrictamente el orden de llegada.

▶ 412 Implementa ahora el tipo *Pila*. Una pila es una lista de elementos en la que el primero que entra es el último en salir. Debes montar los siguientes métodos (además del constructor):

- *apila*: introduce un nuevo elemento en la pila,
- *desapila*: extrae el último elemento introducido en la pila,
- *cima*: devuelve el último elemento introducido en la pila (pero sin modificar la pila),
- *tamaño*: dice cuántos elementos hay apilados,
- *es_vacía*: devuelve cierto si la pila está vacía y falso en caso contrario.

7.3.3. Colas de prioridad

Las colas de prioridad son unas colas un tanto especiales: sus elementos se insertan en cualquier orden, pero el elemento que sale en primer lugar siempre es el que presenta mayor prioridad.

Tenemos varias alternativas para implementar una cola de prioridad[3]:

a) representarla internamente mediante una lista que en todo momento está ordenada de mayor a menor prioridad,

b) representarla internamente mediante una lista desordenada, buscando el elemento de mayor prioridad cada vez que se precise.

Desarrollaremos el segundo caso, pero te proponemos como ejercicio que desarrolles tú mismo el primero.

Las operaciones que podremos realizar con una cola de prioridad son:

- Construirla (*__init__*): crea una cola de prioridad vacía.
- Insertar un elemento (*inserta*): añade un elemento a la cola.
- Consultar el primer elemento (*primero*): devuelve el elemento de mayor prioridad, pero no lo elimina de la cola.
- Extraer (*extrae*): devuelve el elemento de mayor prioridad y lo elimina de la cola.
- Consultar su tamaño (*tamaño*): devuelve el número de elementos encolados.
- Consultar si está vacía (*es_vacía*): devuelve cierto si la cola está vacía y falso en caso contrario.

[3]Con lo que sabemos hacer de momento, solo tenemos esas dos posibilidades. Ambas son muy ineficientes, pero aún es pronto para que sepas por qué. Las colas de prioridad pueden implementarse con *montículos* (*heaps*, en inglés) u otras estructuras de datos avanzadas.

Implementemos:

```
colaprioridad.py
class ColaPrioridad:
    def __init__(self):
        self.cola = []

    def inserta(self, valor):
        self.cola.append(valor)

    def primero(self, valor):
        if len(self.cola) == 0:
            return None
        máximo = self.cola[0]
        for elemento in self.cola:
            if elemento > máximo:
                máximo = elemento
        return máximo

    def extrae(self, valor):
        if len(self.cola) == 0:
            return None
        índice = 0
        for i in range(len(self.cola)):
            if self.cola[i] > self.cola[índice]:
                índice = i
        aux = self.cola[índice]
        del self.cola[índice]
        return aux

    def tamaño(self):
        return len(self.cola)

    def es_vacía(self):
        return len(self.cola) == 0
```

▶ 413 Implementa una cola de prioridad utilizando internamente una lista siempre ordenada.

▶ 414 Vamos a mejorar el programa de gestión de colas de pacientes desarrollado en el apartado anterior. Ahora vamos a clasificar los pacientes según la gravedad de su dolencia en 20 niveles de prioridad distintos. El nivel 20 es el más prioritario, el que requiere la más urgente atención, y el 1 es el menos prioritario.

Podríamos gestionar 20 colas distintas, una para cada prioridad, pero resulta más elegante gestionar una única cola de prioridad. Para ello, en lugar de encolar únicamente el nombre de un paciente, puedes encolar una lista con dos o más datos (como mínimo deberás almacenar la prioridad y el nombre del paciente).

7.3.4. Conjuntos

Es probable que en nuestras aplicaciones necesitemos utilizar conjuntos, es decir, colecciones de datos en las que cada elemento de un universo está o no está presente. Una lista no es un conjunto porque es posible que un elemento aparezca repetidas veces. Podemos simular el tipo de datos conjunto con una simple lista, pero teniendo siempre la precaución de no insertar un elemento si ya está presente. Para no complicar nuestros programas con constantes consultas a listas para determinar si los elementos a insertar están o no ya presentes, es mejor definir un nuevo tipo de datos que, internamente, realice las comprobaciones pertinentes: una clase *Conjunto*. Es más, podemos enriquecer el nuevo tipo de datos con operaciones de conjuntos útiles: la intersección, la unión, la diferencia, etc.

En primer lugar nos hemos de plantear qué atributos tendrá un *Conjunto*. Bastará con una lista que contenga los elementos presentes en el conjunto. Inicialmente, la lista estará vacía:

```
conjunto.py
class Conjunto:
    def __init__(self):
        self.elementos = []
```

Necesitamos definir ahora un método para la inserción de elementos. El método *inserta* recibirá dos argumentos: *self* (como siempre) y el elemento que deseamos insertar. Antes de añadir el elemento a la lista, nos preguntaremos si ya está presente, porque solo deberemos hacer algo en caso contrario:

```
conjunto.py
class Conjunto:
    ...

    def inserta(self, elemento):
        if not (elemento in self.elementos):
            self.elementos.append(elemento)
```

▶ 415 Enriquece la clase *Conjunto* con un método *elimina* que borre del conjunto un elemento dado. (Solo habrá que borrar el elemento de la lista si está presente, claro está).

Definamos ahora un método para imprimir un conjunto por pantalla. Estaría bien que los conjuntos aparecieran por pantalla con el aspecto «tradicional»: con sus elementos separados por comas y encerrados en un par de llaves.

```
conjunto.py
class Conjunto:
    ...

    def __str__(self):
        cadena = '{'
        if len(self.elementos) > 0:
            for elemento in self.elementos[:-1]:
                cadena = cadena + str(elemento) + ', '
            cadena = cadena + str(self.elementos[-1])
        return cadena + '}'
```

He aquí un ejemplo de uso de *Conjunto*:

```
>>> A = Conjunto()
>>> A.inserta(3)
>>> A.inserta(5)
>>> A.inserta(3)
>>> print(A)
{3, 5}
```

Otro método útil nos permite preguntar a un conjunto por su talla, es decir, el número de elementos que lo forman:

```
class Conjunto:
    ...

    def talla(self):
        return len(self.elementos)
```

Nos vendrá bien disponer de un método que permita consultar si un elemento pertenece o no a un conjunto:

```
class Conjunto:
    ...

    def pertenece(self, elemento):
        return elemento in self.elementos
```

▶ 416 Diseña un método *es_vacío* que devuelva cierto si el conjunto está vacío y falso en caso contrario.

Vamos a por las operaciones entre conjuntos. Definamos la unión de conjuntos con un método que devuelva el conjunto resultante de unir al conjunto *self* otro conjunto:

```
class Conjunto:
    ...

    def unión(self, otro):
        C = Conjunto()
        C.elementos = self.elementos[:]
        for elemento in otro.elementos:
            C.inserta(elemento)
        return C
```

Observa cómo se usa *unión*:

```
>>> A = Conjunto()
>>> A.inserta(3)
>>> A.inserta(5)
>>> B = Conjunto()
>>> B.inserta(10)
>>> C = A.unión(B)
>>> print(C)
{3, 5, 10}
```

▶ 417 Diseña un método *intersección* que devuelva un conjunto con la intersección de dos conjuntos (uno de ellos será aquel sobre el que se invoca el método y otro se pasará como parámetro).

▶ 418 Diseña un método *diferencia* que devuelva un conjunto con la diferencia entre dos conjuntos, es decir, con aquellos elementos que están en el primero, pero no en el segundo.

Finalmente, he aquí un método que consulta si otro conjunto dado está incluido en el conjunto (*self*):

```
class Conjunto:
    ...

    def incluye(self, otro):
        for elemento in otro.elementos:
            if not (elemento in self.elementos):
                return False
        return True
```

7.4. Un ejemplo completo: gestión de un videoclub

En este apartado vamos a desarrollar un ejemplo completo y útil usando clases: un programa para gestionar un videoclub. Empezaremos creando la aplicación de gestión para un videoclub básico, muy simplificado, e iremos complicándola poco a poco.

Más métodos especiales

Hemos visto que las clases admiten dos métodos especiales: *__init__* y *__str__*. No son los únicos métodos especiales. Podemos hacer que las clases se comporten de modo similar a los tipos de datos nativos de Python definiendo muchos otros métodos especiales. He aquí unos pocos:

- *__len__*(*self*): Permite aplicar la función predefinida *len* sobre objetos de la clase. Debe devolver la «longitud» o «talla» del objeto. En el caso de colas y conjuntos, por ejemplo, correspondería al número de elementos. Si *A* es un *Conjunto*, podríamos usar *len*(*A*) si antes hubiésemos definido el método *__len__*.
- *__add__*(*self*, *otro*): Permite aplicar el operador de suma (+) a objetos de la clase sobre la que se ha definido. Si, por ejemplo, *A* y *B* son conjuntos, la expresión *C* = *A* + *B* permite asignar al nuevo conjunto *C* la unión de ambos.
- *__mul__*(*self*, *otro*): Permite aplicar el operador de multiplicación (*) a objetos de la clase sobre la que se ha definido.
- *__cmp__*(*self*, *otro*): Permite aplicar los operadores de comparación (<, >, <=, >=, ==, !=) a objetos de una clase. Debe devolver -1 si *self* es menor que *otro*, 0 si son iguales y 1 si *self* es mayor que *otro*.

  Podemos, por ejemplo, definir *__cmp__* en *Persona* para que devuelva -1 cuando la edad *self*.*edad* es menor que *otro*.*edad*, 0 si son iguales y 1 si *self*.*edad* es mayor que *otro*.*edad*. Si *juan* y *pedro* son personas, podremos compararlas con expresiones como *juan* < *pedro* o *juan* != *pedro*.

Consulta la documentación de Python si quieres conocer todos los métodos especiales que puedes definir en tus clases. Tus programas pueden ganar mucho en elegancia si defines los métodos apropiados para cada clase.

7.4.1. Videoclub básico

El videoclub tiene un listado de socios. Cada socio tiene una serie de datos:

- dni,
- nombre,
- teléfono,
- domicilio.

Por otra parte, disponemos de una serie de películas. De cada película nos interesa:

- título,
- género (acción, comedia, musical, etc.).

Supondremos que en nuestro videoclub básico solo hay un ejemplar de cada película.

Empecemos definiendo los tipos básicos con sus métodos especiales: el constructor *__init__* y el conversor a cadena *__str__*:

videoclub.py
```
class Socio:
    def __init__(self, dni, nombre, teléfono, domicilio):
        self.dni = dni
        self.nombre = nombre
        self.teléfono = teléfono
        self.domicilio = domicilio

    def __str__(self):
        return 'DNI: {0}\nNombre: {1}\nTeléfono: {2}\nDomicilio: {3}\n' \
               .format(self.dni, self.nombre, self.teléfono, self.domicilio)

```

```
class Película:
    def __init__(self, título, género):
        self.título = título
        self.género = género

    def __str__(self):
        return 'Título: {0}\nGénero: {1}\n'.format(self.título, self.género)
```

Podemos definir también una clase *Videoclub* que mantenga y gestione las listas de socios y películas:

```
videoclub.py
class Videoclub:
    def __init__(self):
        self.socios = []
        self.películas = []
```

Nuestra aplicación presentará un menú con diferentes opciones. Empecemos por implementar las más sencillas: dar de alta/baja a un socio, dar de alta/baja una película. La función *menú* mostrará el menú de operaciones y leerá la opción que seleccione el usuario de la aplicación. Nuestra primera versión será esta:

```
videoclub.py
def menú():
    print('*** VIDEOCLUB ***')
    print('1) Dar de alta nuevo socio')
    print('2) Dar de baja un socio')
    print('3) Dar de alta nueva película')
    print('4) Dar de baja una película')
    print('5) Salir')
    opción = int(input('Escoge opción: '))
    while opción < 1 or opción > 5:
        opción = int(input('Escoge opción (entre 1 y 5): '))
    return opción
```

En una variable *videoclub* tendremos una instancia de la clase *Videoclub*, y es ahí donde almacenaremos la información del videoclub. Nuestra primera versión del programa presentará este aspecto:

```
videoclub.py
videoclub = Videoclub()

opción = menú()
while opción != 5:

    if opción == 1:
        print('Alta de socio')
        socio = nuevo_socio()
        if videoclub.contiene_socio(socio.dni):
            print('Ya existía un socio con DNI', dni)
        else:
            videoclub.alta_socio(socio)

    elif opción == 2:
        print('Baja de socio')
        dni = input('DNI: ')
        if videoclub.contiene_socio(dni):
            videoclub.baja_socio(dni)
            print('Socio con DNI', dni, 'dado de baja')
        else:
```

```
            print('No existe ningún socio con DNI', dni)

    elif opción == 3:
        print('Alta de película')
        película = nueva_película()
        if videoclub.contiene_película(película.título):
            print('Ya hay una película con título', película.título)
        else:
            videoclub.alta_película(película)

    elif opción == 4:
        print('Baja de película')
        título = input('Título: ')
        if videoclub.contiene_película(título):
            videoclub.baja_película(título)
        else:
            print('No existe ninguna película llamada', título)


opción = menú()
```

Analicémoslo por partes. Empecemos por el fragmento de código que corresponde al alta de un socio. Lo primero que hacemos es pedir la creación de un nuevo socio mediante la función *nuevo_socio*. Esta función leerá de teclado los datos. Definámosla:

```
videoclub.py
def nuevo_socio():
    dni = input('DNI: ')
    nombre = input('Nombre: ')
    teléfono = input('Teléfono: ')
    domicilio = input('Domicilio: ')
    return Socio(dni, nombre, teléfono, domicilio)
```

El socio devuelto por *nuevo_socio* puede haber sido dado de alta previamente en el videoclub, con lo que no sería procedente darlo de alta ahora. A continuación se «pregunta» al videoclub si ya tiene algún socio con el DNI del nuevo socio. Si es así, se muestra un aviso por pantalla, y si no, se da de alta al socio. Observa que se usan dos métodos del videoclub: *contiene_socio*, que recibe un DNI y devuelve cierto o falso, y *alta_socio*, que recibe un socio y lo añade a su lista de socios. Su definición sería:

```
videoclub.py
class Videoclub:
    ...

    def contiene_socio(self, dni):
        for socio in self.socios:
            if socio.dni == dni:
                return True
        return False

    def alta_socio(self, socio):
        self.socios.append(socio)
```

Estudiemos ahora el fragmento de código para dar de baja a un socio. En primer lugar, se pide su DNI. Si el socio existe (lo que se averigua con el método *contiene_socio*, definido justo antes), se le da de baja llamando al método *baja_socio*, que recibe el DNI. Si ningún socio tiene el DNI suministrado, se advierte al usuario del programa con un aviso. Hemos de definir, pues, *baja_socio*:

```
videoclub.py
class Videoclub:
    ...

    def baja_socio(self, dni):
        for i in range(len(self.socios)):
            if self.socios[i].dni == dni:
                del self.socios[i]
                break
```

▶ 419 Define tú mismo los procedimientos que dan de alta/baja una película.

De poca utilidad será el programa si no permite alquilar las películas. ¿Cómo haremos para representar que una película está alquilada a un socio? Tenemos (al menos) dos posibilidades:

- añadir un atributo a cada *Socio* indicando qué película tiene en alquiler (y si no tiene ninguna, su valor será None, por ejemplo),
- añadir un atributo a cada *Película* indicando a quién está alquilada (y si no está alquilada, su valor será None, por ejemplo).

Parece mejor la segunda opción: una operación que realizaremos con frecuencia es preguntar si una película está alquilada o no; por contra, preguntar si un socio tiene o no películas alquiladas parece una operación menos frecuente.

Así pues, tendremos que modificar la definición de la clase *Película*:

```
videoclub.py
class Película:
    def __init__(self, título, género):
        self.título = título
        self.género = género
        self.alquilada = None

    def __str__(self):
        cadena = 'Título:␣{0}\nGénero:␣{1}\n'.format(self.título, self.género)
        if self.alquilada == None:
            cadena = cadena + 'Disponible\n'
        else:
            cadena = cadena + 'Alquilada␣a:␣{0}\n'.format(self.alquilada)
        return cadena
```

Observa que el atributo *alquilada* no se pasa como parámetro a *__init__*. La razón es muy simple: cuando «construimos» una nueva película no está alquilada a nadie, así que el atributo *alquilada* siempre empieza valiendo None. ¿Para qué pasar como argumento a *__init__* un valor que no aporta información alguna?

Añadamos ahora un método que permita alquilar una película (dado su título) a un socio (dado su DNI). La llamada a este método se asociará a la opción 5 del menú, y el final de ejecución de la aplicación se asociará ahora a la opción 6.

```
videoclub.py
...

videoclub = Videoclub()

opción = menú()

while opción != 6:

    if opción == 1:
```

```
        ...

    elif opción == 5:
        print('Alquiler de película')
        título = input('Título de la película: ')
        dni = input('DNI del socio: ')
        hay_película = videoclub.contiene_película(título)
        hay_socio = videoclub.contiene_socio(dni)
        if hay_película and hay_socio:
            videoclub.alquilar_película(título, dni)
        else:
            if not hay_película:
                print('No hay película títulada', título)
            if not hay_socio:
                print('No hay socio con DNI', dni)
    opción = menú()
```

Diseñemos el método *alquilar_película*. Supondremos que existe una película cuyo título corresponde al que nos indican y que existe un socio cuyo DNI es igual al que nos pasan como argumento, pues ambas comprobaciones se efectúan antes de llamar al método.

```
videoclub.py
class Videoclub:
    ...

    def alquilar_película(self, título, dni):
        for película in self.películas:
            if película.título == título and película.alquilada == None:
                película.alquilada = dni
                break
```

En principio, ya está. El método *alquilar_película* recorre la lista de películas del videoclub y solo efectúa el alquiler cuando encuentra una con el título que nos dan y esta está disponible. Pero podemos mejorarlo: el método no nos informa de si finalmente alquiló o no la película en cuestión, lo que hace que no podamos informar al usuario de si la operación se realizó con éxito o no. Vamos a modificarlo para que devuelva cierto si alquiló efectivamente la película, y falso en caso contrario.

```
videoclub.py
class Videoclub:
    ...

    def alquilar_película(self, título, dni):
        for película in self.películas:
            if película.título == título:
                if película.alquilada == None:
                    película.alquilada = dni
                    return True
                else:
                    return False
```

Ahora podemos modificar las acciones asociadas a la opción 5:

```
videoclub.py
videoclub = Videoclub()

opción = menú()
while opción != 6:

    if opción == 1:
```

```
        ...

    elif opción == 5:
        título = input('Título de la película: ')
        dni = input('DNI del socio: ')
        hay_película = videoclub.contiene_película(título)
        hay_socio = videoclub.contiene_socio(dni)
        if hay_película and hay_socio:
            if videoclub.alquilar_película(título, dni):
                print('Operación realizada')
            else:
                print('La película no está disponible')
        else:
            if not hay_película:
                print('No hay película títulada', título)
            if not hay_socio:
                print('No hay socio con DNI', dni)
    opción = menú()
```

▶ 420 Añade una nueva funcionalidad al programa: la devolución de una película alquilada. Diseña para ello un método *devolver_película* que, dado el título de la película, devuelve True si estaba alquilada y False en caso contrario. Además, si estaba alquilada, el método la marcará como disponible (pondrá a None el valor del campo *alquilada*).

A continuación, añade una opción al menú para devolver una película. Las acciones asociadas son:

- pedir el nombre de la película;
- si no existe una película con ese título, dar el aviso pertinente y no hacer nada más;
- si existe la película pero no estaba alquilada, avisar al usuario y no hacer nada más;
- y si existe la película y estaba alquilada, marcarla como disponible.

▶ 421 Modifica los métodos que dan de baja a un socio o una película para que no se permita dar de baja una película alquilada ni a un socio que tiene alguna película en alquiler.

Si no fue posible dar de baja el socio o la película, el método correspondiente devolverá False. Si, por el contrario, se pudo dar de baja a uno u otro, devolverá True.

Modifica a continuación las acciones asociadas a las respectivas opciones del menú para que den los avisos pertinentes en caso de que no sea posible dar de baja a un socio o una película.

Finalmente, vamos a ofrecer la posibilidad de efectuar una consulta interesante a la colección de películas del videoclub. Es posible que un cliente nos pida que le recomendemos películas disponibles dado el género que a él le gusta. Un método de *Videoclub* permitirá obtener este tipo de listados.

videoclub.py

```
class Videoclub:
    ...

    def listado_por_género(self, género):
        for película in self.películas:
            if película.género == género and película.alquilada == None:
                print(título)
```

Solo resta añadir una opción de menú que pida el género para el que solicitamos el listado e invoque al método *listado_por_género*.

▶ 422 Modifica *listado_por_género* para que muestre todas las películas del videoclub, pero indicando al lado del título si está alquilada o disponible.

7.4.2. Un videoclub más realista

El programa que hemos hecho presenta ciertos inconvenientes por su simplicidad: por ejemplo, asume que solo existe un ejemplar de cada película y, al no llevar registro de las fechas de alquiler, permite que un socio alquile una película un número indeterminado de días. Mejoremos el programa corrigiendo ambos defectos.

Tratemos en primer lugar la cuestión de la existencia de varios ejemplares por película. Está claro que la clase *Película* ha de sufrir algunos cambios. Tenemos (entre otras) dos posibilidades:

a) hacer que cada objeto de la clase *Película* corresponda a un ejemplar, es decir, permitir que la lista *películas* contenga títulos repetidos (una vez por cada ejemplar).

b) enriquecer cada película con un campo *ejemplares* que indique cuántos ejemplares tenemos.

Mmmm. La segunda posibilidad requiere un estudio más detallado. Con solo un contador de ejemplares no es suficiente. ¿Cómo representaremos el hecho de que, por ejemplo, de 5 ejemplares, 3 están alquilados, cada uno a un socio diferente? Será preciso enriquecer la información propia de una *Película* con una lista que contenga un elemento por cada ejemplar alquilado. Cada elemento de la lista deberá contener, como mínimo, algún dato que identifique al socio al que se alquiló la película.

Parece, pues, que la primera posibilidad es más sencilla de implementar. Desarrollaremos esa, pero te proponemos como ejercicio que desarrolles tú la segunda posibilidad.

En primer lugar, modificaremos el método que da de alta una película para que nos pida el número de ejemplares que añadimos al videoclub.

```
videoclub.py
class Videoclub:
    ...

    def alta_película(self, película, ejemplares):
        for i in range(ejemplares):
            nuevo_ejemplar = Película(película.título, película.género)
            self.películas.append(nuevo_ejemplar)
```

Al dar de alta ejemplares de una película ya no será necesario comprobar si existe ese título en nuestra colección de películas:

```
videoclub.py
...

    elif opción == 3:
        print('Alta␣de␣película')
        película = nueva_película()
        ejemplares = int(input('Ejemplares:␣'))
        videoclub.alta_película(película, ejemplares)
    ...
```

Dar de baja un número de ejemplares de un título determinado no es muy difícil, aunque puede aparecer una pequeña complicación: que no podamos eliminar efectivamente el número de ejemplares solicitado, bien porque no hay tantos en el videoclub, bien porque alguno de ellos está alquilado. Haremos que el método que da de baja el número de ejemplares solicitado nos devuelva el número de ejemplares que realmente pudo dar de baja, de ese modo al menos «avisamos» a quien nos llama de lo que realmente hicimos.

```
videoclub.py
class Videoclub:
    ...

    def baja_película(self, título, ejemplares):
        bajas_efectivas = 0
```

```
        i = 0
        while i < len(self.películas):
            if self.películas[i].título == título and self.películas[i].alquilada == None:
                del películas[i]
                bajas_efectivas += 1
            else:
                i += 1
        return bajas_efectivas
```

Veamos cómo queda el fragmento de código asociado a la acción de menú que da de baja películas:

videoclub.py
```
...

    elif opción == 4:
        print('Baja de película')
        título = input('Título: ')
        ejemplares = int(input('Ejemplares: '))
        bajas = videoclub.baja_película(título, ejemplares)
        if bajas < ejemplares:
            print('Solo pude dar de baja', bajas, 'ejemplares')
        else:
            print('Operación realizada')

    ...
```

El método de alquiler de una película a un socio necesita una pequeña modificación: puede que los primeros ejemplares encontrados de una película estén alquilados, pero no estamos seguros de si hay alguno libre hasta haber recorrido la colección entera de películas. El método puede quedar así:

videoclub.py
```
class Videoclub:
    ...

    def alquilar_película(self, título, dni):
        for película in self.películas:
            if película.título == título:
                if película.alquilada == None:
                    película.alquilada = dni
                    return True
        return False
```

Observa que solo devolvemos False cuando hemos recorrido la lista entera de películas sin haber podido encontrar una libre.

▶ 423 Implementa el nuevo método de devolución de películas. Ten en cuenta que necesitarás dos datos: el título de la película y el DNI del socio.

Ahora podemos modificar el programa para que permita controlar si un socio retiene la película más días de los permitidos y, si es así, que nos indique los días de retraso. Enriqueceremos la clase *Película* con nuevos atributos:

- *fecha_alquiler*: contiene la fecha en que se realizó el alquiler.
- *días_permitidos*: número de días de alquiler permitidos.

Parece que ahora hemos de disponer de cierto control sobre las fechas. Afortunadamente ya hemos construido una clase *Fecha* en este mismo capítulo. ¡Utilicémosla!

▶ 424 Modifica el constructor de *Película* para añadir los nuevos atributos. Modifica a continuación *alta_película* para que pida también el valor de *días_permitidos*.

Empezaremos por añadir una variable global, a la que llamaremos *hoy*, que contendrá la fecha actual[4]:

videoclub.py
```
from fecha import Fecha, lee_fecha
...

# Programa principal
hoy = lee_fecha()
```

Cuando alquilemos una película no solo apuntaremos el socio al que la alquilamos, también recordaremos la fecha del alquiler:

videoclub.py
```
class Videoclub:
    ...

    def alquilar_película(self, título, dni):
        for película in self.películas:
            if película.título == título:
                if película.alquilada == None:
                    película.alquilada = dni
                    película.fecha_alquiler = hoy
                    return True
        return False
```

Otro procedimiento afectado por la introducción de fechas es el de devolución de películas. No nos podemos limitar a devolver la película: hemos de comprobar si se incurre en retraso y, por tanto, se debe pagar una multa.

▶ 425 Modifica el método de devolución de películas para que tenga en cuenta la fecha de alquiler y la fecha de devolución. El método devolverá el número de días de retraso. (Supón que nuestra clase *Fecha* dispone de un método *días_trascurridos* que devuelve el número de días transcurridos desde una fecha determinada).

Modifica las acciones asociadas a la opción de menú de devolución de películas para que tenga en cuenta el valor devuelto por el método *devolver_película* y muestre por pantalla el número de días de retraso (si es el caso).

▶ 426 Modifica el método *listado_por_género* para que un mismo título de una película no aparezca más que una vez. Al lado del título aparecerá la cadena `'DISPONIBLE'` si hay al menos un ejemplar disponible y `'NO DISPONIBLE'` si todos los ejemplares están alquilados.

7.4.3. Listado completo

Nos ha salido un programa larguito. Vale la pena que mostremos un listado completo.

videoclub.py
```
#==============================
# videoclub
#==============================
# Programa para la gestión de videoclubs.
#==============================

from fecha import Fecha, lee_fecha
```

[4]Lo natural sería que la fecha actual se fijara automáticamente a partir del reloj del sistema. Puedes hacerlo usando el módulo *time*. Consulta el manual de la librería.

```python

#-------------------------------------------------------------
# Socio
#-------------------------------------------------------------
# Clase para almacenar los datos relativos a un socio.
#-------------------------------------------------------------
class Socio:
    def __init__(self, dni, nombre, teléfono, domicilio):
        self.dni = dni
        self.nombre = nombre
        self.teléfono = teléfono
        self.domicilio = domicilio

    def __str__(self):
        return 'DNI: {0}\nNombre: {1}\nTeléfono: {2}\nDomicilio: {3}\n' \
               .format(self.dni, self.nombre, self.teléfono, self.domicilio)

#-------------------------------------------------------------
# Película
#-------------------------------------------------------------
# Clase para almacenar los datos relativos a un ejemplar de una
# película.
#-------------------------------------------------------------
class Película:
    def __init__(self, título, género, días_permitidos):
        self.título = título
        self.género = género
        self.alquilada = None
        self.fecha_alquiler = None
        self.días_permitidos = días_permitidos

    def __str__(self):
        cadena = 'Título: {0}\nGénero: {1}\n'.format(self.título, self.género)
        if self.alquilada == None:
            cadena = cadena + 'Disponible\n'
        else:
            cadena = cadena + 'Alquilada a: {0}\n'.format(self.alquilada)
        return cadena

#-------------------------------------------------------------
# Videoclub
#-------------------------------------------------------------
# Almacena dos listas: una de socios y otra de películas. Los
# elementos de la primera lista son de la clase Socio, y los de la
# segunda, de la clase Película.
#-------------------------------------------------------------
class Videoclub:
    def __init__(self):
        self.socios = []
        self.películas = []

    def contiene_socio(self, dni):
        # Devuelve True si existe algún socio con DNI dni y False en caso
        # contrario.
        for socio in self.socios:
            if socio.dni == dni:
                return True
        return False

    def contiene_película(self, título):
        # Devuelve True si existe alguna película del título que nos
```

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

```
        # pasan y False en caso contrario.
        for película in self.películas:
            if película.título == título:
                return True
        return False

    def alta_socio(self, socio):
        # Añade un socio a la lista de socios.
        # Requisito: no debe existir ningún socio con el mismo DNI.
        self.socios.append(socio)

    def baja_socio(self, dni):
        # Elimina al socio cuyo DNI es igual a dni.
        # Requisito: debe existir un socio con ese DNI.
        for i in range(len(self.socios)):
            if self.socios[i].dni == dni:
                del self.socios[i]
                break

    def alta_película(self, película, ejemplares):
        # Da de alta un número dado de ejemplares de una película.
        for i in range(ejemplares):
            nuevo_ejemplar = Película(película.título, película.género, \
                                      película.días_permitidos)
            self.películas.append(nuevo_ejemplar)

    def baja_película(self, título, ejemplares):
        # Da de baja un número de ejemplares de la película cuyo título nos
        # suministran como argumento. Devuelve el número de ejemplares que
        # se dio de baja efectivamente.
        bajas_efectivas = 0
        i = 0
        while i < len(self.películas) and bajas_efectivas < ejemplares:
            if self.películas[i].título == título and self.películas[i].alquilada == None:
                del self.películas[i]
                bajas_efectivas += 1
            else:
                i += 1
        return bajas_efectivas

    def alquilar_película(self, título, dni):
        # Alquila un ejemplar de la película cuyo título nos indican, al socio
        # con DNI dni. Si no consigue efectuar el alquiler, devuelve False, y True
        # si lo consigue. La fecha de alquiler se fija automáticamente al día
        # actual.
        # Requisito: debe existir un socio con el DNI suministrado.
        for película in self.películas:
            if película.título == título and película.alquilada == None:
                película.alquilada = dni
                película.fecha_alquiler = hoy
                return True
        return False

    def devolver_película(self, título, dni):
        # Devuelve un ejemplar de la película cuyo título nos indican que
        # estaba alquilada al socio con DNI dni. Devuelve el número de días
        # de retraso, o -1 si ningún ejemplar de la película está alquilado
        # al socio.
        # Requisito: debe existir un socio con el DNI suministrado.
        for película in self.películas:
            if película.título == título and película.alquilada == dni:
```

```
                película.alquilada = None
                días_retraso = película.fecha_alquiler.días_transcurridos(hoy)
                return días_retraso
        return -1

    def listado_por_género(self, género):
        # Muestra un listado de las películas cuyo género es el indicado.
        # Cada título aparece solo una vez. Al lado del título aparece
        # una indicación sobre si hay o no hay ejemplares disponibles para
        # alquiler.
        disponibles = []
        alquiladas = []
        for película in self.películas:
            if película.género == género:
                if película.alquilada == None and not (película.título in disponibles):
                    disponibles.append(película.título)
            if película.alquilada != None and not (película.título in alquiladas):
                alquiladas.append(película.título)
        for título in disponibles:
            print(título, 'DISPONIBLE')
        for título in alquiladas:
            if not (título in disponibles):
                print(título, 'NO␣DISPONIBLE')

#............................................................................
# Funciones
#............................................................................

def menú():
    # Muestra el menú por pantalla y lee una opción de teclado, que es el
    # resultado devuelto.
    # La función se asegura de que la opción leída esté entre 0 y 8.
    print('***␣VIDEOCLUB␣***')
    print('0)␣Fijar␣fecha␣actual')
    print('1)␣Dar␣de␣alta␣nuevo␣socio')
    print('2)␣Dar␣de␣baja␣un␣socio')
    print('3)␣Dar␣de␣alta␣nueva␣película')
    print('4)␣Dar␣de␣baja␣una␣película')
    print('5)␣Alquilar␣película')
    print('6)␣Devolver␣película')
    print('7)␣Listado␣por␣género')
    print('8)␣Salir')

    opción = int(input('Escoge␣opción:␣'))
    while opción < 0 or opción > 8:
        opción = int(input('Escoge␣opción␣(entre␣0␣y␣8):␣'))
    return opción

def nuevo_socio():
    # Pide por teclado los datos de un nuevo socio y devuelve un objeto
    # de la clase Socio.
    dni = input('DNI:␣')
    nombre = input('Nombre:␣')
    teléfono = input('Teléfono:␣')
    domicilio = input('Domicilio:␣')
    return Socio(dni, nombre, teléfono, domicilio)

def nueva_película():
    # Pide por teclado los datos de una nueva película y devuelve un
    # objeto de la clase Película.
    título = input('Título:␣')
```

```
    género = input('Género: ')
    días_permitidos = input('Días permitidos: ')
    return Película(título, género, días_permitidos)

#------------------------------------------------------------
# Programa principal
#------------------------------------------------------------

# Fijar fecha actual
hoy = lee_fecha()

videoclub = Videoclub()

opción = menú()
while opción != 8:

    if opción == 0:
        print('Cambiar fecha actual')
        hoy = lee_fecha()

    elif opción == 1:
        print('Alta de socio')
        socio = nuevo_socio()
        if videoclub.contiene_socio(socio.dni):
            print('Ya existía un socio con DNI', dni)
        else:
            videoclub.alta_socio(socio)

    elif opción == 2:
        print('Baja de socio')
        dni = input('DNI: ')
        if videoclub.contiene_socio(dni):
            videoclub.baja_socio( dni )
            print('Socio con DNI', dni, 'dado de baja')
        else:
            print('No existe ningún socio con DNI', dni)

    elif opción == 3:
        print('Alta de película')
        película = nueva_película()
        ejemplares = int(input('Ejemplares: '))
        videoclub.alta_película(película, ejemplares)

    elif opción == 4:
        print('Baja de película')
        título = input('Título: ')
        ejemplares = int(input('Ejemplares: '))
        bajas = videoclub.baja_película(título, ejemplares)
        if bajas < ejemplares:
            print('Solo pude dar de baja', bajas, 'ejemplares')
        else:
            print('Operación realizada')

    elif opción == 5:
        print('Alquilar película')
        título= input('Título de la película: ')
        dni = input('DNI del socio: ')
        hay_película = videoclub.contiene_película(título)
        hay_socio = videoclub.contiene_socio(dni)
        if hay_película and hay_socio:
            if videoclub.alquilar_película(título, dni):
```

```
                print('Operación␣realizada')
            else:
                print('La␣película␣no␣está␣disponible')
        else:
            if not hay_película:
                print('No␣hay␣película␣títulada', título)
            if not hay_socio:
                print('No␣hay␣socio␣con␣DNI', dni)

    elif opción == 6:
        print('Devolver␣película')
        título= input('Título␣de␣la␣película:␣')
        dni = input('DNI␣del␣socio:␣')
        hay_película = videoclub.contiene_película(título)
        hay_socio = videoclub.contiene_socio(dni)
        if hay_película and hay_socio:
            resultado = videoclub.devolver_película(título, dni)
            if resultado == 0:
                print('Operación␣realizada')
            elif resultado > 0:
                print('Película␣entregada␣con␣un␣retraso␣de', resultado, 'días')
            else:
                print('La␣película', título, 'no␣está␣alquilada␣al␣socio', dni)
        else:
            if not hay_película:
                print('No␣hay␣película␣títulada', título)
            if not hay_socio:
                print('No␣hay␣socio␣con␣DNI', dni)

    elif opción == 7:
        print('Listado␣por␣género')
        género = input('Género:␣')
        videoclub.listado_por_género(género)

    opción = menú()
```

El programa de gestión de un videoclub que hemos desarrollado dista de ser perfecto. Muchas de las operaciones que hemos implementado son ineficientes y, además, mantiene toda la información en memoria RAM, así que pierde toda la información al finalizar la ejecución. Tendremos que esperar al próximo capítulo para abordar el problema del almacenamiento de información de modo que «recuerde» su estado entre diferentes ejecuciones.

Bases de datos

Muchos programas de gestión manejan grandes volúmenes de datos. Es posible diseñar programas como el del videoclub (con almacenamiento de datos en disco duro, eso sí) que gestionen adecuadamente la información, pero, en general, poco recomendable. Existen programas y lenguajes de programación orientados a la gestión de bases de datos. Estos sistemas se encargan de gestionar el almacenamiento de información en disco y ofrecen utilidades para acceder y modificar la información. Es posible expresar, por ejemplo, órdenes como «busca todas las películas cuyo género es *acción*» o «lista a todos los socios que llevan un retraso de 1 o más días».

El lenguaje de programación más extendido para consultas a bases de datos es SQL (Standard Query Language) y numerosos sistemas de bases de datos lo soportan. Existen, además, sistemas de bases de datos de distribución gratuita como MySQL o Postgres, suficientemente potentes para aplicaciones de pequeño y mediano tamaño.

En otras asignaturas de la titulación aprenderás a utilizar sistemas de bases de datos y a diseñar bases de datos.

7.4.4. Extensiones propuestas

Te proponemos como ejercicios una serie de extensiones al programa:

▶ 427 Modifica el programa para permitir que una película sea clasificada en diferentes géneros. (El atributo *género* será una lista de cadenas, y no una cadena).

▶ 428 Modifica la aplicación para permitir reservar películas a socios. Cuando de una película no se disponga de ningún ejemplar libre, los socios podrán solicitar una reserva.

¡Ojo!, la reserva se hace sobre una película, no sobre un ejemplar, es decir, la lista de espera de *Matrix* permite a un socio alquilar el primer ejemplar de *Matrix* que quede disponible. Por otra parte, si hay, por ejemplo, dos socios con una misma película reservada, solo podrá alquilarse a otros socios cuando haya tres o más ejemplares libres.

▶ 429 Modifica el programa del ejercicio anterior para que las reservas caduquen automáticamente a los dos días. Es decir, si el socio no ha alquilado la película a los dos días, su reserva expira.

▶ 430 Modifica el programa para que registre el número de veces que se ha alquilado cada película. Mediante una nueva opción de menú, el programa mostrará la lista de las 10 películas más alquiladas hasta el momento.

▶ 431 Modifica el programa para que registre todas las películas que ha alquilado cada socio a lo largo de su vida. Al consultar los datos de un socio se mostrarán sus géneros favoritos.

▶ 432 Añade al programa una opción de menú para aconsejar al cliente. Basándose en su historial de alquileres, el programa determinará sus géneros favoritos y mostrará un listado con las películas de dichos géneros disponibles para alquiler.

7.4.5. Algunas reflexiones

Hemos desarrollado un ejemplo bastante completo, pero lo hemos hecho poco a poco, incrementalmente. Hemos empezado por construir una aplicación para un videoclub básico y hemos ido añadiéndole funcionalidad paso a paso. Normalmente no se desarrollan programas de ese modo. Se parte de una *especificación* de la aplicación, es decir, se parte de una descripción de lo que debe hacer el programa. El programador efectúa un *análisis* de la aplicación a construir. Un buen punto de partida es determinar las *estructuras de datos* que utilizará. En nuestro caso, hemos definido dos clases, *Socio* y *Película*, y hemos decidido que mantendríamos una lista de socios y otra de películas como atributos de otra clase: *Videoclub*. Solo cuando se ha decidido qué estructuras de datos utilizar se está en condiciones de diseñar e implementar el programa.

Pero ahí no acaba el trabajo del programador. La aplicación debe ser *testeada* para, en la medida de lo posible, asegurarse de que no contiene errores. Solo cuando se está seguro de que no los tiene, la aplicación pasa a la fase de *explotación*. Y es probable (¡o seguro!) que entonces descubramos nuevos errores de programación. Empieza entonces un *ciclo de detección y corrección de errores*.

Tras un período de explotación de la aplicación es frecuente que el usuario solicite la implementación de nuevas funcionalidades. Es preciso, entonces, proponer una nueva especificación (o ampliar la ya existente), efectuar su correspondiente análisis e implementar las nuevas características. De este modo llegamos a la producción de nuevas *versiones* del programa.

▶ 433 Nos gustaría retomar el programa de gestión de `MP3` que desarrollamos en el ejercicio 400. Nos gustaría introducir el concepto de «álbum». Cada álbum tiene un título, unos intérpretes y una lista de canciones (ficheros `MP3`). Modifica el programa para que gestione álbumes. Deberás permitir que el usuario dé de alta y baja álbumes, así como que obtenga listados completos de los álbumes disponibles, listados ordenados por intérpretes, búsquedas de canciones en la base de datos, etc.

▶ 434 Deseamos gestionar una biblioteca. La biblioteca contiene libros que los socios pueden tomar prestados un número de días. De cada libro nos interesa, al menos, su título, autor y año de edición. De cada socio mantenemos su DNI, su nombre y su teléfono. Un socio puede tomar prestados tres libros. Si un libro tarda más de 10 días en ser devuelto, el socio no podrá sacar nuevos libros durante un período de tiempo: tres días de penalización por cada día de retraso.

Diseña un programa que permita dar de alta y baja libros y socios y llevar control de los préstamos y devoluciones de los libros. Cuando un socio sea penalizado, el programa indicará por pantalla hasta qué fecha está penalizado e impedirá que efectúe nuevos préstamos hasta entonces.

7.5. Diccionarios

Cambiamos radicalmente de tercio. En este apartado vamos a presentar una nueva estructura de datos de Python: el *diccionario*. Un diccionario Python no es más que una correspondencia entre *claves* y *valores*. Es fácil establecer una relación con los diccionarios a los que estás acostumbrado. Un diccionario de lengua española, por ejemplo, pone en correspondencia palabras (claves) con su definición o lista de definiciones (valores). Un diccionario español-inglés pone en correspondencia palabras españolas (claves) con palabras inglesas (valores). Pero hay más entidades del mundo real que podemos asimilar a diccionarios Python: un listín telefónico, por ejemplo, pone en correspondencia nombres (claves) con números de teléfono (valores); una agenda pone en correspondencia fechas (claves) con citas (valores); el directorio de unos grandes almacenes pone en correspondencia secciones como «moda caballero», «electrónica», «música», etc. (claves) con plantas del edificio (valores); *Google* pone en correspondencia palabras aparecidas en páginas web (claves) con la URL de dichas páginas (valores)... Todas estas entidades y muchas más pueden modelarse en nuestros programas con diccionarios.

7.5.1. Creación de diccionarios

Empezamos por el diccionario más sencillo de todos: el diccionario vacío. El diccionario vacío no pone en correspondencia a nada con nada, así que no resulta demasiado útil... de momento. El diccionario vacío se denota con un par de llaves:

```
>>> d = {}↵
```

Ya tenemos un diccionario vacío en la variable *d*. Utilicémoslo para construir un listín telefónico:

```
>>> d['Juan']  = '964␣37␣64␣32'↵
>>> d['Luis']  = '964␣73␣46␣23'↵
>>> d['Ana']   = '96␣287␣98␣99'↵
>>> d['María'] = '964␣22␣10␣00'↵
```

Una buena noticia: la notación nos resulta familiar, pues es parecida a la de las listas. La diferencia más notable es que *en un diccionario no es preciso que los índices sean números enteros en un rango determinado*, basta con que sean valores de algún tipo inmutable (cadenas, enteros, flotantes)[5].

7.5.2. Consulta en diccionarios

Si deseamos consultar un teléfono, solo tenemos que indexar por el nombre:

```
>>> print(d['Juan'])↵
964 37 64 32
>>> print(d['María'])↵
964 22 10 00
```

¿Y si accedemos a un elemento inexistente?

```
>>> print(d['Pedro'])↵
```

[5]... o tuplas, que son listas inmutables. Las tuplas son listas que se abren y cierran con paréntesis, no con corchetes. Si quieres saber más sobre tuplas, consulta la documentación de Python.

```
Traceback (most recent call last):
  File "<input>", line 1, in <module>
KeyError: 'Pedro'
```

Ya está: se produce un error del tipo *KeyError* (error de clave) y nos especifica que la clave errónea es **'Pedro'**.

Pero si cada vez que accedemos a una clave errónea el programa falla, los diccionarios resultarán un tanto problemáticos. Está todo previsto: el operador **in** te permite preguntar si una clave determinada aparece en un diccionario:

```
>>> print('Juan' in d)↵
True
>>> print('Pedro' in d)↵
False
```

Antes de acceder a un diccionario con una clave «dudosa», deberemos asegurarnos de que existe.

7.5.3. Recorrido de diccionarios

Pronto nos veremos en la necesidad de recorrer todos los elementos de un diccionario para, por ejemplo, mostrarlos en la pantalla. ¿Funcionará el bucle **for-in** con diccionarios al igual que lo hacía con listas?

```
>>> for elemento in d:↵
...     print(elemento)↵
... ↵
Luis
Juan
María
Ana
```

¡Casi! El bucle recorre todas las claves que tenemos en el diccionario. Por lo tanto, a partir de las claves, nos resultará muy fácil acceder también a sus valores:

```
>>> for clave in d:↵
...     print(clave, '->', d[clave])↵
... ↵
Luis -> 964 73 46 23
Juan -> 964 37 64 32
María -> 964 22 10 00
Ana -> 96 287 98 99
```

¡Ya está! Por otra parte, en los diccionarios también disponemos de un método llamado *keys* que nos proporciona un generador de las claves de un diccionario (similar a la función *range* que vimos en capítulos anteriores). De este modo, el recorrido anterior lo podríamos haber escrito de manera equivalente así:

```
>>> for clave in d.keys():↵
...     print(clave, '->', d[clave])↵
... ↵
Luis -> 964 73 46 23
Juan -> 964 37 64 32
María -> 964 22 10 00
Ana -> 96 287 98 99
```

A partir del generador de claves, podemos obtener una lista con todas las claves del diccionario utilizando la función *list*:

```
>>> list(d.keys())↵
['Luis', 'Juan', 'María', 'Ana']
```

Observa que la lista que devuelve el método *keys* no está ordenada de ningún modo: ni alfabéticamente ni por el orden en que asignaste a cada clave su valor. Esta es una propiedad importante de los diccionarios y que debes tener presente: los diccionarios forman un conjunto *desordenado* de correspondencias clave-valor.

7.5.4. Borrado de elementos

Supón que deseamos borrar un elemento del diccionario. Cuando borrábamos elementos de listas utilizábamos **del**. Veamos si funciona con diccionarios:

```
>>> del d['María']↵
>>> for clave in d.keys():↵
...     print(clave, '->', d[clave])↵
... ↵
Luis -> 964 73 46 23
Juan -> 964 37 64 32
Ana -> 96 287 98 99
```

¡Sí! Borrar elementos no requiere que aprendamos, pues, nuevas sentencias o métodos.

7.5.5. Una aplicación: un listín telefónico

Construyamos un programa que gestione un listín telefónico que permita asociar a una persona más de un teléfono. A través de un menú podremos seleccionar diferentes acciones: añadir teléfonos al listín, consultar el listín y eliminar teléfonos del listín.

Mantendremos la agenda en una variable global *listín*. Esa variable será un diccionario cuyas claves son los nombres de las personas y cuyos valores son *listas de cadenas*, así podremos guardar más de un teléfono por persona (cada cadena de la lista será un teléfono).

Las diferentes acciones se implementarán mediante funciones. El programa principal repetirá el proceso de mostrar un menú, leer la opción, leer los datos necesarios para ejecutar la acción y llamar a la función correspondiente.

Ten en cuenta que asociar un teléfono a un nombre no consiste en asignar algo directamente a la clave correspondiente en *listín*: debes preguntar previamente si ya hay teléfonos asociados a ese nombre y, en tal caso, añadir a la lista de teléfonos el nuevo; si no existe el nombre, entonces sí asignaremos algo a la clave, pero ese algo será una *lista* con el teléfono.

listin.py
```
def añadir(listín, nombre, teléfono):
    if nombre in listín:
        if not teléfono in listín[nombre]:
            listín[nombre].append(teléfono)
    else:
        listín[nombre] = [ teléfono ]

def consultar(listín, nombre):
    if nombre in listín:
        return listín[nombre]
    else:
        return []

def eliminar(listín, nombre):
    if nombre in listín:
        del listín[nombre]

def menú():
    opción = 0
    while opción < 1 or opción > 4:
        print('1) Añadir teléfonos')
        print('2) Consultar listín')
        print('3) Eliminar persona del listín')
        print('4) Salir')
        opción = int(input('Escoge opción: '))
    return opción

# Programa principal
listín = {}

opción = 0
```

```
while opción != 4:
    opción = menú()
    if opción == 1:
        nombre = input('Nombre: ')
        teléfono = input('Teléfono: ')
        añadir(listín, nombre, teléfono)
        más = input('¿Deseas añadir otro teléfono a {0} (s/n)?: ' \
                    .format(nombre))
        while más == 's':
            teléfono = input('Teléfono: ')
            añadir(listín, nombre, teléfono)
            más = input('¿Deseas añadir otro teléfono a {0} (s/n)?: ' \
                        .format(nombre))
    elif opción == 2:
        nombre = input('Nombre: ')
        teléfonos = consultar(listín, nombre)
        for teléfono in teléfonos:
            print(teléfono)
    elif opción == 3:
        nombre = input('Nombre: ')
        eliminar(listín, nombre)
```

▶ 435 Diseña un procedimiento que muestre el contenido completo del listín, pero ordenado alfabéticamente. (Puedes usar el método *sort* sobre una lista para ordenarla).

Ya está. Bien, pero vamos a hacerlo más elegante. ¿Por qué no definimos una clase *Listín* que «sepa» añadir, consultar y borrar nombres y teléfonos. Los objetos de la clase *Listín* tendrán un solo atributo: un diccionario.

listin.py

```
class Listín:
    def __init__(self):
        self.listín = {}

    def añadir(self, nombre, teléfono):
        if nombre in self.listín:
            if not teléfono in self.listín[nombre]:
                self.listín[nombre].append(teléfono)
        else:
            self.listín[nombre] = [ teléfono ]

    def consultar(self, nombre):
        if nombre in self.listín:
            return self.listín[nombre]
        else:
            return []

    def eliminar(self, nombre):
        if nombre in self.listín:
            del self.listín[nombre]
# Fin de la clase

def menú():
    opción = 0
    while opción < 1 or opción > 4:
        print('1) Añadir teléfonos')
        print('2) Consultar listín')
        print('3) Eliminar persona del listín')
        print('4) Salir')
        opción = int(input('Escoge opción: '))
    return opción
```

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

```

# Programa principal
listín = Listín()

opción = 0
while opción != 4:
    opción = menú()
    if opción == 1:
        nombre = input('Nombre: ')
        teléfono = input('Teléfono: ')
        listín.añadir(nombre, teléfono)
        más = input('¿Deseas añadir otro teléfono a {0} (s/n)?: ' \
                    .format(nombre))
        while más == 's':
            teléfono = input('Teléfono: ')
            listín.añadir(nombre, teléfono)
            más = input('¿Deseas añadir otro teléfono a {0} (s/n)?: ' \
                        .format(nombre))
    elif opción == 2:
        nombre = input('Nombre: ')
        teléfonos = listín.consultar(nombre)
        for teléfono in teléfonos:
            print(teléfono)
    elif opción == 3:
        nombre = input('Nombre: ')
        listín.eliminar(nombre)
```

¿Qué implementación es mejor? Pues depende. Personalmente, nos parece mejor la segunda: hemos definido un nuevo tipo de datos útil para muchas aplicaciones. Leyendo ambos programas está más claro en el segundo que *añadir*, *consultar* y *eliminar* son métodos asociados a un *listín*. Con clases hemos aumentado la legibilidad.

7.5.6. Un contador de palabras

Los diccionarios tienen muchos usos inesperados. Por ejemplo, nos puede interesar saber cuántas veces aparece cada una de las palabras en un determinado texto, que vendrá dado mediante una serie de líneas que terminará con una línea vacía.

Para ello, utilizaremos un diccionario indexado por palabras. Cada vez que veamos una nueva palabra, asociaremos el valor 1 a la palabra en cuestión y cuando volvamos a ver esa palabra, incrementaremos su valor en una unidad.

contar_veces_palabras.py

```
contador = {}

print('Ve introduciendo líneas (línea vacía para acabar)')
línea = input('Línea: ')
while línea != '':
    palabras = línea.split()
    for palabra in palabras:
        if palabra in contador:
            contador[palabra] += 1
        else:
            contador[palabra] = 1
    línea = input('Línea: ')

# Obtenemos la lista de palabras diferentes
palabras = list(contador.keys())
# Ordenamos la lista de palabras
palabras.sort()
# Recorremos la lista ordenada para mostrar cada contador
print('Se han encontrado las siguientes palabras:')
```

```
for palabra in palabras:
    print('{0} ({1})'.format(palabra, contador[palabra]))
```

Probémoslo para un ejemplo:

```
Ve introduciendo líneas (línea vacía para acabar)
Línea: me han dicho que te han dicho un dicho que yo he dicho
Línea: mas ese dicho que te han dicho que yo he dicho
Línea: no lo he dicho yo
Línea: pues si yo lo hubiera dicho
Línea: estaría bien dicho
Línea: por haberlo dicho yo
Línea: 
Se han encontrado las siguientes palabras:
bien (1)
dicho (11)
ese (1)
estaría (1)
haberlo (1)
han (3)
he (3)
hubiera (1)
lo (2)
mas (1)
me (1)
no (1)
por (1)
pues (1)
que (4)
si (1)
te (2)
un (1)
yo (5)
```

▶ 436 Modifica el programa anterior para que no distinga entre mayúsculas y minúsculas.

▶ 437 Modifica el programa anterior para que, al final, nos diga cuál es la palabra o palabras que aparecen con mayor frecuencia (o sea, la moda).

▶ 438 Diseña una función que, dada una lista de números enteros, calcule su moda. Utiliza diccionarios de un modo similar al del problema que hemos resuelto en este apartado.

▶ 439 Implementa un programa que pida un texto en español y lo traduzca palabra a palabra al inglés. Utiliza un diccionario de Python para almacenar cada palabra con su correspondiente traducción. El programa empezará sin palabra alguna en el diccionario y pidiendo el texto que desea traducir el usuario. A continuación, procederá a traducir palabra por palabra el texto. Cada vez que vea una palabra desconocida, pedirá al usuario su traducción al inglés y la memorizará. Si vuelve a aparecer esa misma palabra, ya no pedirá su traducción, sino que usará la que le dimos previamente. Es obvio que, con este método, la calidad de la traducción no será muy buena que digamos.

7.5.7. Rediseño del programa del videoclub con diccionarios

Ahora que sabemos utilizar diccionarios podemos plantearnos un rediseño del programa del videoclub (en su versión más realista). En nuestra implementación anterior hemos utilizado sendas listas para mantener los socios y las películas del videoclub. La gestión de las listas resulta un tanto farragosa e ineficiente: cada vez que hemos buscado un socio o una película (y prácticamente todas las opciones del menú de la aplicación obligan a ello), hemos tenido que efectuar un recorrido de una de las dos listas. El programa será tanto más ineficiente cuantos más socios y películas tengamos. Los diccionarios pueden ayudarnos porque permiten saber si contienen una clave (el DNI de un socio o el título de una película, por ejemplo) y, en su caso, localizarla instantáneamente.

Nos conviene, pues, que la «lista» de socios sea un diccionario de socios indexado por sus DNI y que la «lista» de películas sea un diccionario de películas indexado por el título:

```
videoclub.py
...

class Videoclub:
    def __init__(self):
        socios = {}
        películas = {}

...
```

Veamos cómo definir ahora los métodos de alta y baja de socios:

```
videoclub.py
class Videoclub:
...

    def alta_socio(self, socio):
        self.socios[socio.dni] = socio

    def baja_socio(self, dni):
        del self.socios[dni]

...
```

El diccionario de películas requiere un examen más detallado. Recuerda que podemos tener más de un ejemplar por película. Lo que haremos es mantener una lista de ejemplares asociada a cada título de película (la clave en el diccionario). Cada ejemplar será un objeto de la clase *Película*.

```
videoclub.py
class Videoclub:
...

    def alta_película(self, película, ejemplares):
        for i in range(ejemplares):
            nuevo_ejemplar = Película(película.título, película.género, \
                                      película.días_permitidos)
            if película.título in self.películas:
                self.películas[película.título].append(nuevo_ejemplar)
            else:
                self.películas[película.título] = [ nuevo_ejemplar ]

...
```

▶ 440 Define el método para dar de baja una cantidad de ejemplares de una película. El método recibirá el título de la película y el número de ejemplares a retirar.

Y acabaremos la exposición mostrándote el método que alquila una película a un socio.

```
videoclub.py
class Videoclub:
...

    def alquilar_película(self, título, dni):
        for película in self.películas[título]:
            if película.alquilada == None:
                película.alquilada = dni
```

```
                película.fecha_alquiler = hoy
                return True
        return False

...
```

▶ 441 Acaba de implementar el programa `videoclub.py` utilizando diccionarios.

Capítulo 8

Ficheros

—Pero, ¿qué dijo el Lirón? —preguntó uno de los miembros del jurado.
—No me acuerdo —dijo el Sombrerero.
—*Tienes* que acordarte —comentó el Rey—; si no, serás ejecutado.

Alicia en el país de las maravillas, Lewis Carroll

Todos los programas que hemos desarrollado hasta el momento empiezan su ejecución en estado de *tabula rasa*, es decir, con la memoria «en blanco». Esto hace inútiles los programas que manejan sus propias bases de datos, como el de gestión de un videoclub desarrollado en el capítulo anterior, pues cada vez que salimos de la aplicación, el programa olvida todos los datos relativos a socios y películas que hemos introducido. Podríamos pensar que basta con no salir nunca de la aplicación para que el programa sea útil, pero salir o no de la aplicación está fuera de nuestro control: la ejecución del programa puede detenerse por infinidad de motivos, como averías del ordenador, apagones, fallos en nuestro programa que abortan su ejecución, operaciones de mantenimiento del sistema informático, etc. La mayoría de los lenguajes de programación permiten almacenar y recuperar información de *ficheros*, esto es, conjuntos de datos residentes en sistemas de almacenamiento secundario (disco duro, disquete, cinta magnética, etc.) que mantienen la información aun cuando el ordenador se apaga.

Un tipo de fichero de particular interés es el que se conoce como *fichero de texto*. Un fichero de texto contiene una sucesión de caracteres que podemos considerar organizada en una secuencia de líneas. Los programas Python, por ejemplo, suelen residir en ficheros de texto. Es posible generar, leer y modificar ficheros de texto con editores de texto o con nuestros propios programas[1]. En este capítulo solo estudiaremos ficheros de texto.

8.1. Generalidades sobre ficheros

Aunque puede que ya conozcas lo suficiente sobre los sistemas de ficheros, no estará de más que repasemos brevemente algunos aspectos fundamentales y fijemos terminología.

8.1.1. Sistemas de ficheros: directorios y ficheros

En los sistemas Unix (como Linux) hay una única estructura de *directorios* y *ficheros*. Un fichero es una agrupación de datos y un directorio es una colección de ficheros y/u otros directorios (atento a la definición recursiva). El hecho de que un directorio incluya ficheros y otros directorios determina una relación jerárquica entre ellos. El nivel más alto de la jerarquía es la *raíz*, que se denota con una barra «`/`» y es un directorio. Es usual que la raíz contenga un directorio llamado `home` (hogar) en el que reside el directorio principal de cada uno de los usuarios del sistema. El directorio principal de cada usuario se llama del mismo modo que su nombre en clave (su `login`).

[1] Editores de texto como XEmacs, por ejemplo, escriben y leen ficheros de texto. En Microsoft Windows puedes usar el bloc de notas para generar ficheros de texto.

En la figura 8.1 se muestra un sistema de ficheros Unix como el que hay montado en el servidor de la Universitat Jaume I (los directorios se representan enmarcados con un recuadro). El primer directorio, la raíz, se ha denotado con `/`. En dicho directorio está ubicado, entre otros, el directorio `home`, que cuenta con un subdirectorio para cada usuario del sistema. Cada directorio de usuario tiene el mismo nombre que el `login` del usuario correspondiente. En la figura puedes ver que el usuario `al55555` tiene dos directorios (`practicas` y `trabajos`) y un fichero (`nota.txt`). Es usual que los nombres de fichero tengan dos partes separadas por un punto. En el ejemplo, `nota.txt` se considera formado por el nombre propiamente dicho, `nota`, y la *extensión*, `txt`. No es obligatorio que los ficheros tengan extensión, pero sí conveniente. Mediante la extensión podemos saber fácilmente de qué `tipo` es la información almacenada en el fichero. Por ejemplo, `nota.txt` es un fichero que contiene texto, sin más, pues el convenio seguido es que la extensión `txt` está reservada para ficheros de texto. Otras extensiones comunes son: `py` para programas Python[2], `c` para programas C[3], `html` o `htm` para ficheros HTML[4], `pdf` para ficheros PDF[5], `mp3` para ficheros de audio en formato MP3[6], `ps` para ficheros Postscript[7], `jpg` o `jpeg` para fotografías comprimidas con pérdida de calidad…

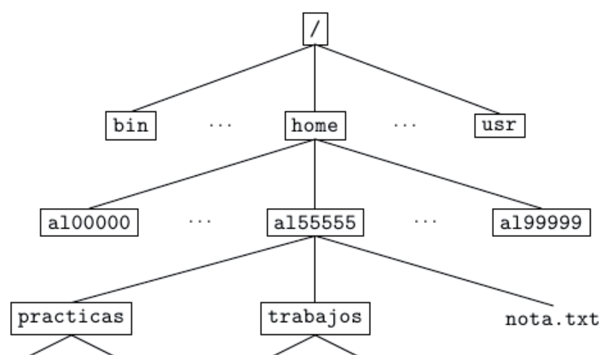


Figura 8.1: Un sistema de ficheros Unix.

8.1.2. Rutas

Es posible que en el sistema de ficheros haya dos o más ficheros con el mismo nombre. Si es el caso, estos ficheros estarán en directorios diferentes. Todo fichero o directorio es identificado de forma única por su *ruta* (en inglés, «path»), es decir, por el nombre precedido de una descripción del lugar en el que reside siguiendo un «camino» en la jerarquía de directorios.

Cada elemento de una ruta se separa del siguiente mediante una barra. Por ejemplo, la ruta `/home/al55555` consta de dos elementos: el directorio `home`, ubicado en la raíz, y el directorio `al55555`, ubicado dentro del directorio `home`. Es la ruta del *directorio personal* (o *principal*) del usuario `al55555`. El fichero `nota.txt` que reside en ese directorio tiene por ruta `/home/al55555/nota.txt`.

En principio, debes proporcionar la ruta completa (desde la raíz) hasta un fichero para acceder a él, pero no siempre es así. En cada instante «estás» en un directorio determinado: el llamado *directorio activo*. Cuando accedes a un sistema Unix con tu nombre en clave y contraseña, tu directorio activo es tu directorio personal (por ejemplo, en el caso del usuario `al55555`, el directorio `/home/al55555`) (véase figura 8.2). Puedes cambiar de directorio activo con el comando `cd` (abreviatura en inglés de «change directory»). Para acceder a ficheros del

[2]Los programas Python también son ficheros de texto, pero especiales en tanto que pueden ser ejecutados mediante un intérprete de Python.

[3]También los programas C son ficheros de texto, traducibles a código de máquina con un compilador de C.

[4]Nuevamente ficheros de texto, pero visualizables mediante navegadores *web*.

[5]Un formato de texto visualizable con ciertas aplicaciones. Se utiliza para impresión de alta calidad y creación de documentos multimedia. Es un formato definido por la empresa Adobe.

[6]Formato *binario*, es decir, no de texto, en el que hay audio comprimido con pérdida de calidad. Es un formato comercial definido por la empresa Fraunhofer-Gesellschaft.

[7]Fichero de texto con un programa en lenguaje PostScript, de Adobe, que describe una o varias páginas impresas.

directorio activo no es necesario que especifiques rutas completas: basta con que proporciones el nombre del fichero. Es más, si deseas acceder a un fichero que se encuentra en algún directorio del directorio activo, basta con que especifiques únicamente el nombre del directorio y, separado por una barra, el del fichero. En la siguiente figura hemos destacado el directorio activo con un trazo grueso. Desde el directorio activo, `/home/al55555`, la ruta `trabajos/nota.txt` hace referencia al fichero `/home/al55555/trabajos/nota.txt`. Y `nota.txt` también es una ruta: la que accede al fichero `/home/al55555/nota.txt`.

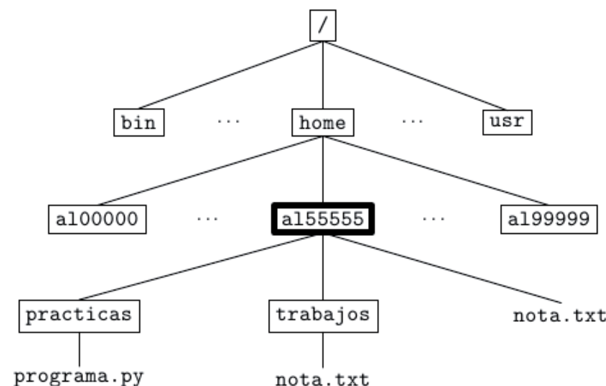


Figura 8.2: Directorio activo por defecto al iniciar una sesión Unix (destacado con trazo grueso).

El directorio padre de un directorio, es decir, el directorio que lo contiene, se puede denotar con dos puntos seguidos (`..`). Así, desde el directorio principal de un usuario, `..` es equivalente a `/home`. Puedes utilizar `..` en rutas absolutas o relativas. Por ejemplo, `/home/al55555/..` también es equivalente a `/home`, pues se refiere al padre del directorio `/home/al55555`. Por otra parte, la ruta `/home/al99999/../al55555/nota.txt` se refiere al mismo fichero que la ruta `/home/al55555/nota.txt`, ¿ves por qué? Finalmente, el propio directorio activo tiene también un nombre abreviado: un punto. Por ejemplo, `./nota.txt` es equivalente a `nota.txt`.

Si una ruta no empieza con la barra, se dice que es *relativa* y «empieza» en el directorio activo, no en la raíz; por contraposición, las rutas cuyo primer carácter es una barra se denominan *absolutas*.

8.1.3. Montaje de unidades

Los diferentes dispositivos de almacenamiento secundario (CD-ROM, DVD, disquetes, memorias Compact-Flash, etc.) se deben *montar* en el sistema de ficheros antes de ser usados. Al montar una unidad, se informa al sistema operativo de que el dispositivo está conectado y se desea acceder a su información. El acceso a sus ficheros y directorios se efectúa a través de las rutas adecuadas. En Unix, es típico que cada dispositivo se monte como un subdirectorio de `/mnt`[8]. Por ejemplo, `/mnt/floppy` suele ser el disquete («floppy disk», en inglés), `/mnt/cdrom` el CD-ROM y `/mnt/cdrecorder` la grabadora de discos compactos.

Para montar una unidad debes ejecutar el comando `mount` seguido del directorio que corresponde a dicha unidad (siempre que tengas permiso para hacerlo). Por ejemplo, `mount /mnt/floppy` monta la disquetera. Si has montado con éxito la unidad, se puede acceder a su contenido con rutas que empiezan por `/mnt/floppy`. Como el disquete tiene sus propios directorios y ficheros, la ruta con la que accedes a su información usa el prefijo `/mnt/floppy/`, va seguida de la secuencia de directorios «dentro» del disquete y acaba con el nombre del fichero (o directorio). Para acceder a un fichero `mio.txt` en un directorio del disquete llamado `miscosas` y que ya ha sido montado, has de usar la ruta `/mnt/floppy/miscosas/mio.txt`.

Una vez has dejado de usar una unidad, puedes desmontarla con el comando `umount` seguido de la ruta al correspondiente directorio. Puedes desmontar el disquete, por ejemplo, con `umount /mnt/floppy`.

[8]Pero solo eso: típico. En algunos sistemas, los dispositivos se montan directamente en el directorio raíz; en otros, en un directorio llamado `/media`.

Peculiaridades del sistema de ficheros de Microsoft Windows

En Microsoft Windows las cosas son un poco más complicadas. Por una parte, el separador de elementos de la ruta es la barra *invertida* «\». Como la barra invertida es el carácter con el que se inician las secuencias de escape, debes tener especial cuidado al utilizarlo en una cadena. La ruta `\directorio\fichero.txt`, por ejemplo, se codificará en una cadena Python como `'\\directorio\\fichero.txt'`. Por otra parte, existen diferentes *volúmenes* o *unidades*, cada uno de ellos con su propia raíz y directorio activo. En lugar de montar cada dispositivo en un directorio del sistema de ficheros, Microsoft Windows le asigna una letra y una raíz propias. Típicamente, la letra `A` corresponde a la disquetera y la letra `C` al disco duro principal (pero ni siquiera eso es seguro).

Cuando deseamos dar una ruta absoluta hemos de indicar en primer lugar la unidad separada por dos puntos del resto de la ruta. Por ejemplo `D:\practicas\programa.py` hace referencia al fichero `programa.py` que se encuentra en el directorio `practicas` de la raíz de la unidad `D` (probablemente un disco duro).

Dado que hay más de un directorio activo a la vez, hay también una *unidad activa*. Cuando das una ruta relativa sin indicar letra de unidad, se toma como punto de partida el directorio activo de la unidad activa. Si usas una ruta relativa precedida de una letra de unidad y dos puntos, partirás del directorio activo de dicha unidad. Si usas una ruta absoluta pero no especificas letra de unidad, se entiende que partes de la raíz de la unidad activa.

8.2. Ficheros de texto

Ya estamos en condiciones de empezar a trabajar con ficheros de texto. Empezaremos por la lectura de ficheros de texto. Los ficheros con los que ilustraremos la exposición puedes crearlos con cualquier editor de texto (XEmacs, o `vi` en Unix; el Bloc de Notas en Microsoft Windows).

8.2.1. El protocolo de trabajo con ficheros: abrir, leer/escribir, cerrar

Desde el punto de vista de la programación, los ficheros son objetos en los que podemos escribir y/o leer información. El trabajo con ficheros obliga a seguir *siempre* un protocolo de tres pasos:

a) *Abrir* el fichero indicando su ruta (relativa o absoluta) y el modo de trabajo. Hay varios modos de trabajo:

   - *Lectura*: es posible leer información del fichero, pero no modificarla ni añadir nueva información.
   - *Escritura*: solo es posible escribir información en el fichero. Por regla general, la apertura de un fichero en modo escritura borra todo el contenido previo del mismo.
   - *Lectura/escritura*: permite leer y escribir información del fichero.
   - *Adición*: permite añadir nueva información al fichero, pero no modificar la ya existente.

b) *Leer* o *escribir* la información que desees.

c) *Cerrar* el fichero.

Es importante que sigas siempre estos tres pasos. Es particularmente probable que olvides cerrar el fichero, pues Python no detectará esta circunstancia como un fallo del programa. Aún así, no cerrar un fichero se considera un grave error de programación. Lee el cuadro «¿Y por qué hay que cerrar los ficheros?» si quieres saber por qué.

8.2.2. Lectura de ficheros de texto línea a línea

Empecemos por un ejemplo completo: un programa que muestra el contenido de un fichero de texto. Fíjate en este programa:

visualiza.py
```
# Paso 1: abrir el fichero.
```

¿Y por qué hay que cerrar los ficheros?

Una vez has acabado de trabajar con un fichero, siempre debes cerrarlo. No podemos enfatizar suficientemente lo importante que es cerrar todos los ficheros tan pronto hayas acabado de trabajar con ellos, especialmente si los has modificado. *Si no cierras el fichero, es posible que los cambios que hayas efectuado se pierdan o, peor aún, que el fichero se corrompa.*

Hay razones técnicas para que sea así. El trabajo con sistemas de almacenamiento secundario (discos duros, disquetes, discos compactos, etc.) es, en principio, muy ineficiente, al menos si lo comparamos con el trabajo con memoria RAM. Los dispositivos de almacenamiento secundario suelen tener componentes mecánicos y su manejo es mucho más lento que el de los componentes puramente electrónicos. Para leer/escribir un dato en un disco duro, por ejemplo, lo primero que ha de hacer el sistema es desplazar el brazo con el cabezal de lectura/escritura hasta la pista que contiene la información; a continuación, debe esperar a que el sector que contiene ese dato pase por debajo del cabezal; solo entonces se podrá leer/escribir la información. Ten en cuenta que estas operaciones requieren, en promedio, *mili*segundos, cuando los accesos a memoria RAM tardan *nano*segundos, una diferencia de velocidad del orden de ¡un millón! Pagar un coste tan alto por cada acceso a un dato residente en disco duro haría prácticamente imposible trabajar con él.

El sistema operativo se encarga de hacer eficiente el uso de estos dispositivos utilizando *buffers* («tampones», en español). Un buffer es una memoria intermedia (usualmente residente en RAM). Cuando leemos un dato del disco duro, el sistema operativo no lleva a memoria solo ese dato, sino muchos otros que están próximos a él (en su mismo sector, por ejemplo). ¿Por qué? Porque cabe esperar razonablemente que próximas lecturas tengan lugar sobre los datos que siguen al que acabamos de leer. Ten en cuenta que leer estos otros datos es rápido, pues con la lectura del primero ya habíamos logrado poner el cabezal del disco sobre la pista y sector correspondientes. Así, aunque solo pidamos leer en un instante dado un byte (un carácter), el sistema operativo lleva a memoria, por ejemplo, cuatro kilobytes. Esta operación se efectúa de forma transparente para el programador y evita que posteriores lecturas accedan realmente al disco.

La técnica de uso de buffers también se utiliza al escribir datos en el fichero. Las operaciones de escritura se realizan en primera instancia sobre un buffer, y no directamente sobre el disco. Solo en determinadas circunstancias, como la saturación del buffer o el cierre del fichero, se escribe efectivamente en el disco duro el contenido del buffer.

Y llegamos por fin a la importancia de cerrar el fichero. Cuando das la orden de cierre de un fichero, estás haciendo que se vuelque el buffer en el disco duro y que se libere la memoria que ocupaba. Si un programa finaliza accidentalmente sin que se haya volcado el buffer, los últimos cambios se perderán o, peor aún, el contenido del fichero se corromperá haciéndolo ilegible. Probablemente, más de una vez habrás experimentado problemas de este tipo como mero usuario de un sistema informático: al quedarse colgado el ordenador con una aplicación abierta, se ha perdido el documento sobre el que estabas trabajando.

El beneficio de cerrar convenientemente el fichero es, pues, doble: por un lado, te estás asegurando de que los cambios efectuados en el fichero se registren definitivamente en el disco duro y, por otro, se libera la memoria RAM que ocupa el buffer.

Recuérdalo: abrir, trabajar… *y cerrar* siempre.

```
2 fichero = open('ejemplo.txt', 'r')
3
4 # Paso 2: leer los datos del fichero.
5 for línea in fichero:
6     print(línea)
7
8 # Paso 3: cerrar el fichero.
9 fichero.close()
```

Analicémoslo paso a paso. La segunda línea abre el fichero (en inglés, «open» significa abrir). Observa que *open* es una función que recibe dos argumentos (ambos de tipo cadena): el *nombre del fichero* (su ruta), que en este ejemplo es relativa, y el *modo de apertura*. En el ejemplo hemos abierto un fichero llamado `ejemplo.txt` en *modo de lectura* (la letra `r` es abreviatura de «read», que en inglés significa leer). Si abrimos un fichero en modo de lectura, solo podemos leer su contenido, pero no modificarlo. La función *open* devuelve un objeto que almacenamos en la variable *fichero*. Toda operación que efectuemos sobre el fichero se hará a través del

Precauciones al trabajar con ficheros

Te hemos insistido mucho en que debes cerrar todos los ficheros tan pronto hayas acabado de trabajar con ellos. Si la aplicación finaliza normalmente, el sistema operativo cierra todos los ficheros abiertos, así que no hay pérdida de información. Esto es bueno y malo a la vez. Bueno porque si olvidas cerrar un fichero y tu programa está, por lo demás, correctamente escrito, al salir todo quedará correctamente almacenado; y malo porque es fácil que te relajes al programar y olvides la importancia que tiene el correcto cierre de los ficheros. Esta falta de disciplina hará que acabes por no cerrar los ficheros cuando hayas finalizado de trabajar con ellos, pues «ellos solos ya se cierran al final». Una invitación al desastre.

El riesgo de pérdida de información inherente al trabajo con ficheros hace que debas ser especialmente cuidadoso al trabajar con ellos. Es deseable que los ficheros permanezcan abiertos el menor intervalo de tiempo posible. Si una función o procedimiento actúa sobre un fichero, esa subrutina debería abrir el fichero, efectuar las operaciones de lectura/escritura pertinentes y cerrar el fichero. Solo cuando la eficiencia del programa se vea seriamente comprometida, deberás considerar otras posibilidades.

Es más, deberías tener una política de copias de seguridad para los ficheros de modo que, si alguna vez se corrompe uno, puedas volver a una versión anterior tan reciente como sea posible.

identificador *fichero*. Al abrir un fichero para lectura, Python comprueba si el fichero existe. Si no existe, el intérprete de Python aborta la ejecución y nos advierte del error. Si ejecutásemos ahora el programa, sin un fichero `ejemplo.txt` en el directorio activo, obtendríamos un mensaje similar a este:

```
Traceback (most recent call last):
  File "visualiza.py", line 2, in <module>
    fichero = open('ejemplo.txt', 'r')
IOError: [Errno 2] No such file or directory: 'ejemplo.txt'
```

Se ha generado una excepción del tipo *IOError* (abreviatura de «input/output error»), es decir, un error de entrada/salida.

Tratamiento de errores al trabajar con ficheros

Si tratas de abrir en modo lectura un fichero inexistente, obtienes un error y la ejecución del programa aborta. Tienes dos posibilidades para reaccionar a esta eventualidad y evitar el fin de ejecución del programa. Una consiste en preguntar antes si el fichero existe:

visualiza.py
```
1 import os
2
3 if os.path.exists('ejemplo.txt'):
4     fichero = open('ejemplo.txt', 'r')
5     for línea in fichero:
6         print(línea)
7     fichero.close()
8 else:
9     print('El fichero no existe.')
```

La otra pasa por capturar la excepción que genera el intento de apertura:

visualiza.py
```
1 try:
2     fichero = open('ejemplo.txt', 'r')
3     for línea in fichero:
4         print(línea)
5     fichero.close()
6 except IOError:
7     print('El fichero no existe.')
```

Si todo ha ido bien, el bucle de la línea 5 recorrerá el contenido del fichero línea a línea.

Para cada línea del fichero, pues, se mostrará el contenido por pantalla. Finalmente, en la línea 9 (ya fuera del bucle) se cierra el fichero con el método *close* (que en inglés significa cerrar). A partir de ese instante, está prohibido efectuar nuevas operaciones sobre el fichero. El único modo en que podemos volver a leer el fichero es abriéndolo de nuevo.

Hagamos una prueba. Crea un fichero llamado `ejemplo.txt` con el editor de texto y guárdalo en el mismo directorio en el que has guardado `visualiza.py`. El contenido del fichero debe ser este:

ejemplo.txt
```
Esto es
un ejemplo de texto almacenado
en un fichero de texto.
```

Ejecutemos el programa, a ver qué ocurre:

```
Esto es

un ejemplo de texto almacenado

en un fichero de texto.
```

Algo no ha ido bien del todo: ¡hay una línea en blanco tras cada línea leída! La explicación es sencilla: las líneas finalizan en el fichero con un salto de línea (carácter `\n`) y la cadena con la línea leída contiene dicho carácter. Por ejemplo, la primera línea del fichero de ejemplo es la cadena `'Esto␣es\n'`. Al hacer *print* de esa cadena, aparecen en pantalla dos saltos de línea: el que corresponde a la visualización del carácter `\n` de la cadena, más el propio del *print*.

Si deseamos eliminar esos saltos de línea espúreos, deberemos modificar el programa:

visualiza.py
```
1 fichero = open('ejemplo.txt', 'r')
2
3 for línea in fichero:
4     if línea[-1] == '\n':
5         línea = línea[:-1]
6     print(línea)
7
8 fichero.close()
```

Ahora sí:

```
Esto es
un ejemplo de texto almacenado
en un fichero de texto.
```

Fíjate en que la quinta línea del programa modifica la cadena almacenada en *línea*, pero *no modifica en absoluto el contenido del fichero*. Una vez lees de un fichero, trabajas con una copia en memoria de la información, y no directamente con el fichero.

Desarrollemos ahora otro ejemplo sencillo: un programa que calcula el número de líneas de un fichero de texto. El nombre del fichero de texto deberá introducirse por teclado.

lineas.py
```
nombre = input('Nombre␣del␣fichero:␣')
fichero = open(nombre, 'r')

contador = 0
for línea in fichero:
    contador += 1

fichero.close()

print(contador)
```

Texto y cadenas

Como puedes ver, el resultado de efectuar una lectura sobre un fichero de texto es una cadena. Es muy probable que buena parte de tu trabajo al programar se centre en la manipulación de las cadenas leídas.

Un ejemplo: imagina que te piden que cuentes el número de palabras de un fichero de texto entendiendo que uno o más espacios separan una palabra de otra (no prestaremos atención a los signos de puntuación). El programa será sencillo: abrir el fichero; leer línea a línea y contar cuántas palabras contiene cada línea; y cerrar el fichero. La dificultad estribará en la rutina de cálculo del número de palabras de una línea. Pues bien, recuerda que hay un método sobre cadenas que devuelve una lista con cada una de las palabras que esta contiene: *split*. Si usas *len* sobre la lista devuelta por *split* habrás contado el número de palabras.

Otro método de cadenas muy útil al tratar con ficheros es *strip* (en inglés significa «pelar»), que devuelve una copia sin blancos (espacios, tabuladores o saltos de línea) delante o detrás. Por ejemplo, el resultado de `'␣un␣ejemplo␣\n'.strip()` es la cadena `'un␣ejemplo'`. Dos métodos relacionados son *lstrip*, que elimina los blancos de la izquierda (la «l» inicial es por «left»), y *rstrip*, que elimina los blancos de la derecha (la «r» inicial es por «right»).

▶ 442 Diseña un programa que cuente el número de caracteres de un fichero de texto, incluyendo los saltos de línea. (El nombre del fichero se pide al usuario por teclado).

▶ 443 Haz un programa que, dada una palabra y un nombre de fichero, diga si la palabra aparece o no en el fichero. (El nombre del fichero y la palabra se pedirán al usuario por teclado).

▶ 444 Haz un programa que, dado un nombre de fichero, muestre cada una de sus líneas precedida por su número de línea. (El nombre del fichero se pedirá al usuario por teclado).

▶ 445 Haz una *función* que, dadas la ruta de un fichero y una palabra, devuelva una lista con las líneas que contienen a dicha palabra.

Diseña a continuación un programa que lea el nombre de un fichero y tantas palabras como el usuario desee (utiliza un bucle que pregunte al usuario si desea seguir introduciendo palabras). Para cada palabra, el programa mostrará las líneas que contienen dicha palabra en el fichero.

▶ 446 Haz un programa que muestre por pantalla la línea más larga de un fichero. Si hay más de una línea con la longitud de la más larga, el programa mostrará únicamente la primera de ellas. (El nombre del fichero se pedirá al usuario por teclado).

▶ 447 Haz un programa que muestre por pantalla todas las líneas más largas de un fichero. (El nombre del fichero se pedirá al usuario por teclado). ¿Eres capaz de hacer que el programa lea una sola vez el fichero?

▶ 448 La orden `head` («cabeza», en inglés) de Unix muestra las 10 primeras líneas de un fichero. Haz un programa `head.py` que muestre por pantalla las 10 primeras líneas de un fichero. (El nombre del fichero se pedirá al usuario por teclado).

▶ 449 En realidad, la orden `head` de Unix muestra las `n` primeras líneas de un fichero, donde `n` es un número suministrado por el usuario. Modifica `head.py` para que también pida el valor de `n` y muestre por pantalla las `n` primeras líneas del fichero.

▶ 450 La orden `tail` («cola», en inglés) de Unix muestra las 10 últimas líneas de un fichero. Haz un programa `tail.py` que muestre por pantalla las 10 *últimas* líneas de un fichero. (El nombre del fichero se pide al usuario por teclado). ¿Eres capaz de hacer que tu programa lea una sola vez el fichero? Pista: usa una lista de cadenas que almacene las 10 últimas cadenas que has visto en cada instante.

▶ 451 Modifica `tail.py` para que pida un valor `n` y muestre las `n` últimas líneas del fichero.

▶ 452 El fichero `/etc/passwd` de los sistemas Unix contiene información acerca de los usuarios del sistema. Cada línea del fichero contiene datos sobre un usuario. He aquí una línea de ejemplo:

```
al55555:x:1000:2000:Pedro Pérez:/home/al55555:/bin/bash
```

En la línea aparecen varios campos separados por dos puntos (`:`). El primer campo es el nombre clave del usuario; el segundo *era* la contraseña cifrada (por razones de seguridad, ya no está en `/etc/passwd`); el tercero es su número de usuario (cada usuario tiene un número diferente); el cuarto es su número de grupo (en la UJI, cada titulación tiene un número de grupo); el quinto es el nombre real del usuario; el sexto es la ruta de su directorio principal; y el séptimo es el intérprete de órdenes.

Haz un programa que muestre el nombre de todos los usuarios reales del sistema.

(Nota: recuerda que el método *split* puede serte de gran ayuda).

▶ 453 Haz un programa que pida el nombre clave de un usuario y nos diga su nombre de usuario real utilizando `/etc/passwd`. El programa no debe leer todo el fichero a menos que sea necesario: tan pronto encuentre la información solicitada, debe dejar de leer líneas del fichero.

▶ 454 El fichero `/etc/group` contiene una línea por cada grupo de usuarios del sistema. He aquí una línea de ejemplo:

```
gestion:x:2000:
```

Al igual que en `/etc/passwd`, los diferentes campos aparecen separados por dos puntos. El primer campo es el nombre del grupo; el segundo no se usa; y el tercero es el número de grupo (cada grupo tiene un número diferente).

Haz un programa que solicite al usuario un nombre de grupo. Tras consultar `/etc/group`, el programa listará el nombre real de todos los usuarios de dicho grupo relacionados en el fichero `/etc/passwd`.

▶ 455 El comando `wc` (por «word count», es decir, «conteo de palabras») de Unix cuenta el número de bytes, el número de palabras y el número de líneas de un fichero. Implementa un comando `wc.py` que pida por teclado el nombre de un fichero y muestre por pantalla esos tres datos acerca de él.

8.2.3. Lectura carácter a carácter

No solo es posible leer los ficheros de texto de línea en línea. Podemos leer, por ejemplo, de carácter en carácter. El siguiente programa cuenta el número de caracteres de un fichero de texto:

caracteres.py
```python
nombre = input('Nombre del fichero: ')
fichero = open(nombre, 'r')

contador = 0
carácter = fichero.read(1)
while carácter != '':
    contador += 1
    carácter = fichero.read(1)

fichero.close()
print(contador)
```

El método *read* actúa sobre un fichero abierto y recibe como argumento el número de caracteres que deseamos leer. El resultado es una cadena con, a lo sumo, ese número de caracteres.

Acceso a la línea de órdenes (I)

En los programas que estamos haciendo trabajamos con ficheros cuyo nombre o bien está predeterminado o bien se pide al usuario por teclado durante la ejecución del programa. Imagina que diseñas un programa **cabeza.py** que muestra por pantalla las **n** primeras líneas de un fichero. Puede resultar incómodo de utilizar si, cada vez que lo arrancas, el programa se detiene para pedirte el fichero con el que quieres trabajar y el número de líneas iniciales a mostrar. En los intérpretes de órdenes Unix (y también en el intérprete DOS de Microsoft Windows) hay una forma alternativa de «pasar» información a un programa: proporcionar argumentos en la línea de órdenes. Por ejemplo, podríamos indicar a Python que deseamos ver las 10 primeras líneas de un fichero llamado **texto.txt** escribiendo en la línea de órdenes lo siguiente:

```
$ python3 cabeza.py texto.txt 10
```

¿Cómo podemos hacer que nuestro programa sepa lo que el usuario nos indicó en la línea de órdenes? La variable *argv*, predefinida en *sys*, es una lista que contiene en cada una de sus celdas una de las palabras (como cadena) de la línea de órdenes (excepto la palabra *python3*).

En nuestro ejemplo, el nombre del fichero con el que el usuario quiere trabajar está en *argv*[1] y el número de líneas en *argv*[2] (como cadena). El programa podría empezar así:

opciones_ejecucion.py

```python
from sys import argv

# Obtiene los parámetros de la línea de órdenes
nombre = argv[1]
número = int(argv[2])

# Muestra las primeras líneas del fichero
fichero = open(nombre, 'r')
contador = 0
for línea in fichero:
    print(línea.rstrip())
    contador += 1
    if contador == número:
        break
fichero.close()
```

Cuando se ha llegado al final del fichero y no hay más texto que leer, *read* devuelve la cadena vacía.

El siguiente programa muestra en pantalla una versión cifrada de un fichero de texto. El método de cifrado que usamos es bastante simple: se sustituye cada letra minúscula (del alfabeto inglés) por su siguiente letra, haciendo que a la **z** le suceda la **a**.

cifra.py

```python
nombre = input('Nombre␣del␣fichero:␣')
fichero = open(nombre, 'r')

carácter = fichero.read(1)
while carácter != '':
    if carácter >= 'a' and carácter <= 'y':
        codificado = chr(ord(carácter) + 1)
    elif carácter == 'z':
        codificado = 'a'
    else:
        codificado = carácter
    print(codificado, end='')
    carácter = fichero.read(1)
fichero.close()
```

Acceso a la línea de órdenes (y II)

Usualmente se utiliza una notación especial para indicar los argumentos en la línea de órdenes. Por ejemplo, el número de líneas puede ir precedido por el texto `-n`, de modo que disponemos de cierta libertad a la hora de posicionar los argumentos donde nos convenga:

```
$ python3 cabeza.py texto.txt -n 10
$ python3 cabeza.py -n 10 texto.txt
```

Y si uno de los argumentos, como `-n`, no aparece, se asume un valor por defecto para él (pongamos que el valor 10). Es decir, esta forma de invocar el programa sería equivalente a las dos anteriores:

```
$ python3 cabeza.py texto.txt
```

Un programa que gestiona correctamente esta notación, más libre, podría ser este:

opciones_ejecucion_mas_libre.py

```python
from sys import argv, exit

# Obtiene los parámetros de la línea de órdenes
número = 10
nombre = ''
i = 1
while i < len(argv):
    if argv[i] == '-n':
        i += 1
        if i < len(argv):
            número = int(argv[i])
        else:
            print('Error: en la opción -n no indica valor numérico.')
            exit(0) # La función exit finaliza en el acto la ejecución del programa.
    else:
        if nombre == '':
            nombre = argv[i]
        else:
            print('Error: hay más de un nombre de fichero.')
            exit(0)
    i += 1

... # Muestra las primeras líneas del fichero
```

▶ 456 Haz un programa que lea un fichero de texto que pueda contener vocales acentuadas y muestre por pantalla una versión del mismo en el que cada vocal acentuada haya sido sustituida por la misma vocal sin acentuar.

8.2.4. Otra forma de leer línea a línea

Puede interesarte en ocasiones leer una sola línea de un fichero de texto. Pues bien, el método *readline* hace precisamente eso. Este programa, por ejemplo, lee un fichero línea a línea y las va mostrando por pantalla, pero haciendo uso de *readline*:

linea_a_linea.py

```python
fichero = open('unfichero.txt', 'r')
línea = fichero.readline()
while línea != '':
    print(línea.rstrip())
    línea = fichero.readline()
fichero.close()
```

Observa cuándo finaliza el bucle: al leer la cadena vacía, pues esta indica que hemos llegado al final del fichero.

La abstracción de los ficheros y la *web*

Los ficheros de texto son una poderosa abstracción que encuentra aplicación en otros campos. Por ejemplo, ciertos módulos permiten manejar la World Wide Web como si fuera un inmenso sistema de ficheros. En Python, el módulo *urllib* permite abrir páginas web y leerlas como si fueran ficheros de texto. Este ejemplo te ayudará a entender a qué nos referimos:

```
from urllib.request import urlopen

fichero = urlopen('http://www.uji.es')
for línea in fichero:
    print(línea.decode('utf-8').rstrip())
fichero.close()
```

Salvo por la función de apertura, *urlopen*, no hay mucha diferencia con la lectura de ficheros de texto. En realidad, desde la versión 3 de Python, cada componente del fichero no es realmente una cadena sino una secuencia de bytes. Esto permite manejar fácilmente las posibles diferentes codificaciones de los caracteres. El método *decode* se encarga de convertir la secuencia de bytes leída en una cadena con la codificación indicada (en el ejemplo `utf-8`).

Lectura completa en memoria

Hay un método sobre ficheros que permite cargar todo el contenido del fichero en memoria. Si *f* es un fichero, *f*.*readlines*() lee el fichero completo como *lista de cadenas*. El método *readlines* resulta muy práctico, pero debes usarlo con cautela: si el fichero que lees es muy grande, puede que no quepa en memoria y tu programa, pese a estar «bien» escrito, falle.

También el método *read* puede leer el fichero completo. Si lo usas sin argumentos (*f*.*read*()), el método devuelve *una única cadena* con el contenido íntegro del fichero. Naturalmente, el método *read* presenta el mismo problema que *readlines* si tratas de leer ficheros grandes.

No solo conviene evitar la carga en memoria para evitar problemas con ficheros grandes. En cualquier caso, cargar el contenido del fichero en memoria supone un mayor consumo de la misma y un programador siempre debe procurar no malgastar los recursos del computador.

8.2.5. Escritura de ficheros de texto

Ya estamos en condiciones de aprender a escribir datos en ficheros de texto. Para no cambiar de tercio, seguiremos con el programa de cifrado. En lugar de mostrar por pantalla el texto cifrado, vamos a hacer que `cifra.py` lo almacene en otro fichero de texto:

```
cifra.py
1  nombre_entrada = input('Nombre␣del␣fichero␣de␣entrada:␣')
2  nombre_salida = input('Nombre␣del␣fichero␣de␣salida:␣')
3  fichero_entrada = open(nombre_entrada, 'r')
4  fichero_salida = open(nombre_salida, 'w')
5  carácter = fichero_entrada.read(1)
6  while carácter != '':
7      if carácter >= 'a' and carácter <= 'y':
8          codificado = chr(ord(carácter) + 1)
9      elif carácter == 'z':
10         codificado = 'a'
11     else:
12         codificado = carácter
13     fichero_salida.write(codificado)
14     carácter = fichero_entrada.read(1)
15 fichero_entrada.close()
16 fichero_salida.close()
```

Analicemos los nuevos elementos del programa. En primer lugar (línea 4), el modo en que se abre un fichero para escritura: solo se diferencia de la apertura para lectura en el segundo argumento, que es la cadena `'w'` (abreviatura de «write»). La orden de escritura es *write*, que

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

recibe una cadena y la escribe, sin más, en el fichero (línea 13). La orden de cierre del fichero sigue siendo *close* (línea 16).

No es preciso que escribas la información carácter a carácter. Puedes escribir línea a línea o como quieras. Eso sí, si quieres escribir líneas ¡recuerda añadir el carácter `\n` al final de cada línea!

Esta nueva versión, por ejemplo, lee el fichero línea a línea y lo escribe línea a línea.

```
cifra.py
nombre_entrada = input('Nombre␣del␣fichero␣de␣entrada:␣')
nombre_salida = input('Nombre␣del␣fichero␣de␣salida:␣')

fichero_entrada = open(nombre_entrada, 'r')
fichero_salida = open(nombre_salida, 'w')

for línea in fichero_entrada:
    nueva_línea = ''
    for carácter in línea:
        if carácter >= 'a' and carácter <= 'y':
            codificado = chr(ord(carácter) + 1)
        elif carácter == 'z':
            codificado = 'a'
        else:
            codificado = carácter
        nueva_línea += codificado
    fichero_salida.write(nueva_línea)

fichero_entrada.close()
fichero_salida.close()
```

Los ficheros de texto generados pueden ser abiertos con cualquier editor de textos. Prueba a abrir un fichero cifrado con XEmacs o eclipse (o con el bloc de notas, si trabajas con Microsoft Windows).

▶ 457 Diseña un programa, **descifra.py**, que descifre ficheros cifrados por **cifra.py**. El programa pedirá el nombre del fichero cifrado y el del fichero en el que se guardará el resultado.

▶ 458 Diseña un programa que, dados dos ficheros de texto, nos diga si el primero es una versión cifrada del segundo (con el código de cifrado descrito en la sección).

Aún desarrollaremos un ejemplo más. Empecemos por un programa que genera un fichero de texto con una tabla de números: los números del 1 al 5000 y sus respectivos cuadrados:

```
↯ tabla.py
fichero = open('tabla.txt', 'w')

for i in range(1, 5001):
    fichero.write(i)
    fichero.write(i**2)

fichero.close()
```

Mal: el método *write* solo trabaja con cadenas, no con números. He aquí una versión correcta:

```
tabla.py
fichero = open('tabla.txt', 'w')

for i in range(1, 5001):
    fichero.write(str(i) + '␣' + str(i**2) + '\n')

fichero.close()
```

Y ahora considera esta otra:

```
tabla.py
fichero = open('tabla.txt', 'w')

for i in range(1, 5001):
    fichero.write('{0}␣{1}\n'.format(i, i**2))

fichero.close()
```

Observa lo útil que resulta el método *format* al escribir cadenas formadas a partir de números.

▶ 459 Diseña un programa que obtenga los 100 primeros números primos y los almacene en un fichero de texto llamado `primos.txt`.

▶ 460 Haz un programa que pida el nombre de un grupo de usuarios Unix. A continuación, abre en modo escritura un fichero con el mismo nombre del grupo leído y extensión `grp`. En dicho fichero deberás escribir el nombre real de todos los usuarios de dicho grupo, uno en cada línea. (Lee antes el enunciado de los ejercicios 452 y 454).

▶ 461 Deseamos automatizar el envío personalizado de correo electrónico a nuestros clientes. (¿Recuerdas el apartado 5.1.10? Si no, estúdialo de nuevo). Disponemos de un fichero de clientes llamado `clientes.txt` en el que cada línea tiene la dirección de correo electrónico y el nombre de un cliente nuestro. El fichero empieza así:

```
al00000@alumail.uji.es Pedro Pérez
spammer@spam.com John Doe
...
```

En otro fichero, llamado `carta.txt`, tenemos un carta personalizable. En ella, el lugar donde queremos poner el nombre del cliente aparece marcado con el texto `#CLIENTE#`. La carta empieza así:

```
Estimado/a Sr/a #CLIENTE#:

Tenemos noticias de que ud., don/doña #CLIENTE#, no ha abonado el importe
de la cuota mensual a que le obliga el draconiano contrato que firmó
...
```

Haz un programa que envíe un correo a cada cliente con el contenido de `carta.txt` debidamente personalizado. Ahora que sabes definir y usar funciones, diseña el programa sirviéndote de ellas.

▶ 462 Nuestro fichero `clientes.txt` se modifica ahora para incluir como segundo campo de cada línea el sexo de la persona. La letra `H` indica que se trata de un hombre y la letra `M` que se trata de una mujer. Modifica el programa para que sustituya las expresiones `don/doña` por `don` o `doña`, `Estimado/a` por `Estimado` o `Estimada` y `Sr/a` por `Sr` o `Sra` según convenga.

8.2.6. Añadir texto a un fichero

Has de tener presente que cuando abres un fichero de texto en modo escritura *se borra todo su contenido*. ¿Cómo añadir, pues, información? Una forma trivial es crear un nuevo fichero con una copia del actual, abrir para escritura el original y copiar en él la copia (¡!) para, antes de cerrarlo, añadir la nueva información. Un ejemplo ilustrará mejor la idea. Este programa añade una línea a un fichero de texto:

```
añadir_linea.py
nombre_fichero = input('Fichero:')
```

```
nueva_línea = input('Línea:')
nombre_copia = nombre_fichero + '.copia'

# Hacemos una copia
fichero1 = open(nombre_fichero,'r')
fichero2 = open(nombre_copia, 'w')
for línea in fichero1:
    fichero2.write(línea)
fichero2.close()
fichero1.close()

# y rehacemos el original añadiendo la nueva línea.
fichero1 = open(nombre_copia,'r')
fichero2 = open(nombre_fichero, 'w')
for línea in fichero1:
    fichero2.write(línea)
fichero2.write(nueva_línea + '\n')
fichero2.close()
fichero1.close()
```

El programa presenta bastantes inconvenientes:

- es lento: se leen completamente dos ficheros y también se escriben completamente los datos de los dos ficheros.
- se ha de crear un fichero temporal (si quieres saber qué es un fichero temporal, lee el cuadro titulado «Ficheros temporales y gestión de ficheros y directorios») para mantener la copia del fichero original. El nombre del nuevo fichero puede coincidir con el de otro ya existente, en cuyo caso se borraría su contenido.

Ficheros temporales y gestión de ficheros y directorios

Se denomina fichero temporal a aquel que juega un papel instrumental para llevar a cabo una tarea. Una vez ha finalizado la tarea, los ficheros temporales pueden destruirse sin peligro. El problema de los ficheros temporales es encontrar un nombre de fichero diferente del de cualquier otro fichero existente. El módulo *tempfile* proporciona la función *mktemp*(), que devuelve una cadena correspondiente a la ruta de un fichero que no existe. Si usas esa cadena como nombre del fichero temporal, no hay peligro de que destruyas información. Por regla general, los ficheros temporales se crean en el directorio `/tmp`.

Lo normal es que cuando has cerrado un fichero temporal desees borrarlo completamente. Abrirlo en modo escritura para cerrarlo inmediatamente no es suficiente: si bien el fichero pasa a ocupar 0 bytes (no tiene contenido alguno), sigue existiendo en el sistema de ficheros. Puedes eliminarlo suministrando la ruta del fichero como argumento de la función *remove* (en inglés significa «elimina») del módulo *os*. El módulo *os* contiene otras funciones útiles para gestionar ficheros y directorios. Por citar algunas: *mkdir* crea un directorio, *rmdir* elimina un directorio, *chdir* cambia el directorio activo, *listdir* devuelve una lista con el nombre de todos los ficheros y directorios contenidos en un directorio, y *rename* cambia el nombre de un fichero por otro.

Si solo deseas *añadir* información a un fichero de texto, hay un procedimiento alternativo: abrir el fichero en *modo adición*. Para ello, debes pasar la cadena `'a'` como segundo argumento de *open*. Al abrirlo, no se borrará el contenido del fichero, y cualquier escritura que hagas tendrá lugar al final del mismo.

El siguiente programa de ejemplo pide una «nota» al usuario y la añade a un fichero de texto llamado `notas.txt`.

añadir_nota.py
```
nota = input('Nota a añadir: ')
fichero = open('notas.txt', 'a')
fichero.write(nota + '\n')
```

```
fichero.close()
```

Con cada ejecución de `añadir_nota.py` el fichero `notas.txt` crece en una línea.

8.2.7. Cosas que no se pueden hacer con ficheros de texto

Hay una acción útil que no podemos llevar a cabo con ficheros de texto: posicionarnos directamente en una línea determinada y actuar sobre ella. Puede que nos interese acceder directamente a la, pongamos, quinta línea de un fichero para leerla. Pues bien, la única forma de hacerlo es leyendo las cuatro líneas anteriores, una a una. La razón es simple: cada línea puede tener una longitud diferente, así que no hay ninguna forma de calcular en que posición exacta del fichero empieza una línea cualquiera... a menos, claro está, que leamos las anteriores una a una.

Y otra acción prohibida en los ficheros es el borrado de una línea (o fragmento de texto) cualquiera o su sustitución por otra. Imagina que deseas eliminar un usuario del fichero `/etc/passwd` (y tienes permiso para ello, claro está). Una vez localizado el usuario en cuestión, sería deseable que hubiera una orden «borra-línea» que eliminase esa línea del fichero o «sustituye-línea» que sustituyese esa línea por otra vacía, pero esa orden *no existe*. Ten en cuenta que la información de un fichero se escribe en posiciones contiguas del disco; si eliminaras un fragmento de esa sucesión de datos o lo sustituyeras por otra de tamaño diferente, quedaría un «hueco» en el fichero o machacarías información de las siguientes líneas.

Cuando abras un fichero de texto en Python, elige bien el modo de trabajo: lectura, escritura o adición.

8.2.8. Un par de ficheros especiales: el teclado y la pantalla

Desde el punto de vista de la programación, el teclado es, sencillamente, un fichero más. De hecho, puedes acceder a él a través de una variable predefinida en el módulo *sys*: *stdin* (abreviatura de «standard input», es decir, «entrada estándar»).

El siguiente programa, por ejemplo, solicita que se teclee una línea y muestra por pantalla la cadena leída.

de_teclado.py
```
from sys import stdin

print('Teclea un texto y pulsa retorno de carro')
línea = stdin.readline()
print(línea)
```

Cuando uno pide la lectura de una línea, el programa se bloquea hasta que el usuario escribe un texto y pulsa el retorno de carro. Ten en cuenta que la cadena devuelta incluye un salto de línea al final. La función *input* no es más que una «fachada» para simplificar la lectura de datos del teclado. Puedes considerar que *input* llama primero a *print* si le pasas una cadena y, a continuación, a *stdin.readline*, pero eliminando el salto de línea que este método añade al final de la línea.

Observa que no es necesario «abrir» el teclado (*stdin*) antes de empezar a trabajar con él ni cerrarlo al finalizar. Una excepción, pues, a la regla.

El siguiente programa, por ejemplo, lee de teclado y repite lo que escribimos hasta que «se acabe» el fichero (o sea, el teclado):

eco.py
```
from sys import stdin

for línea in stdin:
    print(línea)
```

Al ejecutar el programa, ¿cómo indicamos que el fichero especial «teclado» acaba? No podemos hacerlo pulsando directamente el retorno de carro, pues en tal caso *línea* tiene información

(el carácter salto de línea) y Python entiende que el fichero aún no ha acabado. Para que el fichero acabe has de introducir una «marca de fin de fichero». En Unix el usuario puede teclear la letra `d` mientras pulsa la tecla de control para indicar que no hay más datos de entrada. En Microsoft Windows se utiliza la combinación `C-z`. Prueba a ejecutar el programa anterior y, cuando desees que termine su ejecución, pulsa `C-d` (o `C-z` si está trabajando con Microsoft Windows) cuando el programa espere leer otra línea.

Otro fichero con el que ya has trabajado es la pantalla. La pantalla es accesible con el identificador *stdout* (abreviatura de «standard output», o sea, «salida estándar») predefinido en el módulo *sys*. Se trata, naturalmente, de un fichero ya abierto en modo de escritura. La sentencia *print* solo es una forma cómoda de usar el método *write* sobre *stdout*, pues añade automáticamente espacios en blanco entre los elementos que separamos con comas y, si procede, añade un salto de línea al final.

8.3. Una aplicación

Es hora de poner en práctica lo aprendido con una pequeña aplicación. Vamos a implementar una sencilla agenda que permita almacenar el nombre y primer apellido de una persona y su teléfono.

La agenda se almacenará en un fichero de texto llamado **agenda.txt** y residente en el directorio activo. Cada entrada de la agenda ocupará tres líneas del fichero, una por cada campo (nombre, apellido y teléfono). He aquí un ejemplo de fichero **agenda.txt**:

```
Antonio
López
964112200
Pedro
Pérez
964001122
```

Presentaremos dos versiones de la aplicación:

- una primera en la que se maneja directamente el fichero de texto,
- y otra en la que el fichero de texto se carga y descarga con cada ejecución.

Vamos con la primera versión. Diseñaremos en primer lugar funciones para las posibles operaciones:

- buscar el teléfono de una persona dados su nombre y apellido,
- añadir una nueva entrada en la agenda,
- borrar una entrada de la agenda dados el nombre y el apellido de la correspondiente persona.

agenda.py
```
def buscar_entrada(nombre, apellido):
    fichero = open('agenda.txt', 'r')
    línea1 = fichero.readline()
    while línea1 != '':
        línea2 = fichero.readline()
        línea3 = fichero.readline()
        if nombre == línea1[:-1] and apellido == línea2[:-1]:
            fichero.close()
            return línea3[:-1]
        línea1 = fichero.readline()
    fichero.close()
    return None

```

```
14 def añadir_entrada(nombre, apellido, teléfono):
15     fichero = open('agenda.txt', 'a')
16     fichero.write(nombre + '\n')
17     fichero.write(apellido + '\n')
18     fichero.write(teléfono + '\n')
19     fichero.close()
20
21 def borrar_entrada(nombre, apellido):
22     fichero = open('agenda.txt', 'r')
23     fcopia = open('agenda.txt.copia', 'w')
24     línea1 = fichero.readline()
25     while línea1 != '':
26         línea2 = fichero.readline()
27         línea3 = fichero.readline()
28         if nombre != línea1[:-1] or apellido != línea2[:-1]:
29             fcopia.write(línea1)
30             fcopia.write(línea2)
31             fcopia.write(línea3)
32         línea1 = fichero.readline()
33     fichero.close()
34     fcopia.close()
35
36     fcopia = open('agenda.txt.copia', 'r')
37     fichero = open('agenda.txt', 'w')
38     for línea in fcopia:
39         fichero.write(línea)
40     fcopia.close()
41     fichero.close()
```

Completa tú mismo la aplicación para que aparezca un menú que permita seleccionar la operación a realizar. Ya lo has hecho varias veces y no ha de resultarte difícil.

▶ 463 Hemos decidido sustituir las tres llamadas al método *write* de las líneas 29, 30 y 31 por una sola:

```
fcopia.write(línea1 + línea2 + línea3)
```

¿Funcionará igual?

▶ 464 En su versión actual, es posible añadir dos veces una misma entrada a la agenda. Modifica *añadir_entrada* para que solo añada una nueva entrada si corresponde a una persona diferente. Añadir por segunda vez los datos de una misma persona supone sustituir el viejo teléfono por el nuevo.

▶ 465 Añade a la agenda las siguientes operaciones:

- Listado completo de la agenda por pantalla. Cada entrada debe ocupar una sóla línea en pantalla.
- Listado de teléfonos de todas las personas cuyo apellido empieza por una letra determinada.

▶ 466 Haz que cada vez que se añade una entrada a la agenda, esta quede ordenada alfabéticamente.

▶ 467 Deseamos poder trabajar con más de un teléfono por persona. Modifica el programa de la agenda para que la línea que contiene el teléfono contenga una relación de teléfonos separados por blancos. He aquí un ejemplo de entrada con tres teléfonos:

```
Pedro
López
964112537 964009923 96411092
```

La función *buscar_entrada* devolverá una lista con tantos elementos como teléfonos tiene la persona encontrada. Enriquece la aplicación con la posibilidad de borrar uno de los teléfonos de una persona.

La segunda versión carga en memoria el contenido completo de la base de datos y la manipula sin acceder a disco. Al finalizar la ejecución, vuelca todo el contenido a disco. Nuestra implementación define una clase para las entradas de la agenda y otra para la propia agenda.

agenda2.py

```python
# Clase Entrada
class Entrada:
    def __init__(self, nombre, apellido, teléfono):
        self.nombre = nombre
        self.apellido = apellido
        self.teléfono = teléfono

def lee_entrada():
    nombre = input('Nombre: ')
    apellido = input('Apellido: ')
    teléfono = input('Teléfono: ')
    return Entrada(nombre, apellido, teléfono)

# Clase Agenda
class Agenda:
    def __init__(self):
        self.lista = []

    def buscar_teléfono(self, nombre, apellido):
        for entrada in self.lista:
            if entrada.nombre == nombre and entrada.apellido == apellido:
                return entrada.teléfono
        return None

    def añadir_entrada(self, entrada):
        self.lista.append(entrada)

    def borrar_entrada(self, nombre, apellido):
        for i in range(len(self.lista)):
            if self.lista[i].nombre == nombre and self.lista[i].apellido == apellido:
                del self.lista[i]
                return


def cargar_agenda():
    agenda = Agenda()
    fichero = open('agenda.txt', 'r')
    línea1 = fichero.readline()
    while línea1 != '':
        línea2 = fichero.readline()
        línea3 = fichero.readline()
        entrada = Entrada(línea1[:-1], línea2[:-1], línea3[:-1])
        agenda.añadir_entrada(entrada)
        línea1 = fichero.readline()
    fichero.close()
    return agenda

def guardar_agenda(agenda):
    fichero = open('agenda.txt', 'w')
```

```
    for entrada in agenda.lista:
        fichero.write(entrada.nombre + '\n')
        fichero.write(entrada.apellido + '\n')
        fichero.write(entrada.teléfono + '\n')
    fichero.close()

# Menú de usuario
def menú():
    print('1) Añadir entrada')
    print('2) Consultar agenda')
    print('3) Borrar entrada')
    print('4) Salir')
    opción = int(input('Seleccione opción: '))
    while opción < 1 or opción > 4:
        opción = int(input('Seleccione opción (entre 1 y 4): '))
    return opción

# Programa principal
agenda = cargar_agenda()

opción = menú()
while opción != 4:
    if opción == 1:
        entrada = lee_entrada()
        agenda.añadir_entrada(entrada)
    elif opción == 2:
        nombre = input('Nombre: ')
        apellido = input('Apellido: ')
        teléfono = agenda.buscar_teléfono(nombre, apellido)
        if teléfono == None:
            print('No está en la agenda')
        else:
            print('Teléfono:', teléfono)
    elif opción == 3:
        nombre = input('Nombre: ')
        apellido = input('Apellido: ')
        agenda.borrar_entrada(nombre, apellido)
    opción = menú()

guardar_agenda(agenda)
```

Esta segunda implementación presenta ventajas e inconvenientes respecto a la primera:

- Al cargar el contenido completo del fichero en memoria, puede que desborde la capacidad del ordenador. Imagina que la agenda ocupa 1 gigabyte en disco duro: será imposible cargarla en memoria en un ordenador de 256 o 512 megabytes.
- El programa solo recurre a leer y escribir datos al principio y al final de su ejecución. Todas las operaciones de adición, edición y borrado de entradas se realizan en memoria, así que su ejecución será *mucho* más rápida.
- Como gestionamos la información en memoria, si el programa aborta su ejecución (por error nuestro o accidentalmente al sufrir un apagón), se pierden todas las modificaciones de la sesión de trabajo actual.

▶ 468 Modifica la aplicación de gestión de estudiantes del capítulo anterior para que recuerde todos los datos entre ejecución y ejecución. (Puedes inspirarte en la segunda versión de la agenda).

Andrés Marzal / Isabel Gracia / Pedro García - ISBN: 978-84-697-1178-1 Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93
Índice

469 Modifica la aplicación de gestión del videoclub del capítulo anterior para que recuerde todos los datos entre ejecución y ejecución. Mantén dos ficheros distintos: uno para las películas y otro para los socios.

8.4. Texto con formato

Un fichero de texto no tiene más que eso, texto; pero ese texto puede escribirse siguiendo una reglas precisas (un formato) y expresar significados inteligibles para ciertos programas. Hablamos entonces de ficheros con formato.

El World Wide Web, por ejemplo, establece un formato para documentos hipertexto: el HTML (HyperText Mark-up Language, o lenguaje de marcado para hipertexto). Un fichero HTML es un fichero de texto cuyo contenido sigue unas reglas precisas. Simplificando un poco, el documento empieza con la *marca* `<HTML>` y finaliza con la marca `</HTML>` (una marca es un fragmento de texto encerrado entre < y >). Entre ellas aparece (entre otros elementos) el par de marcas `<BODY>` y `</BODY>`. El texto se escribe entre estas últimas dos marcas. Cada párrafo empieza con la marca `<P>` y finaliza con la marca `</P>`. Si deseas resaltar un texto con negrita, debes encerrarlo entre las marcas `<B>` y `</B>`, y si quieres destacarlo con cursiva, entre `<I>` y `</I>`. Bueno, no seguimos: ¡la especificación completa del formato HTML nos ocuparía un buen número de páginas! He aquí un ejemplo de fichero HTML:

```
<HTML>
 <BODY>
  <P>
   Un <I>ejemplo</I> de fichero en formato <B>HTML</B> que contiene un par
   de párrafos y una lista:
  </P>
  <OL>
   <LI>Un elemento.</LI>
   <LI>Y uno más.</LI>
  </OL>
  <P><B>HTML</B> es fácil.</P>
 </BODY>
</HTML>
```

Cuando un navegador *web* visualiza una página, está leyendo un fichero de texto y analizando su contenido. Cada marca es interpretada de acuerdo con su significado y produce en pantalla el resultado esperado. Cuando Firefox, Konqueror, Chrome, Internet Explorer o Lynx muestran el fichero `ejemplo.html` interpretan su contenido para producir un resultado visual semejante a este:

> Un *ejemplo* de fichero en formato **HTML** que contiene un par de párrafos y una lista:
>
> - Un elemento.
> - Y uno más.
>
> **HTML** es fácil.

Las ventajas de que las páginas *web* sean meros ficheros de texto (con formato) son múltiples:

- se pueden escribir con cualquier editor de texto,
- se pueden llevar de una máquina a otra sin (excesivos) problemas de portabilidad,
- se pueden manipular con cualquier herramienta de procesado de texto (y hay muchas ya escritas en el entorno Unix),
- *se pueden generar automáticamente desde nuestros propios programas.*

Este último aspecto es particularmente interesante: nos permite crear *aplicaciones web*. Una aplicación web es un programa que atiende peticiones de un usuario (hechas desde una página web con un navegador), consulta bases de datos y muestra las respuestas al usuario formateando la salida como si se tratara de un fichero HTML.

CGI

En muchas aplicaciones se diseñan interfaces para la *web*. Un componente crítico de estas interfaces es la generación automática de páginas *web*, es decir, de (pseudo-)ficheros de texto en formato HTML.

Las aplicaciones *web* más sencillas se diseñan como conjuntos de programas CGI (por «Common Gateway Interface», algo como «Interfaz Común de Pasarela»). Un programa CGI recibe una estructura de datos que pone en correspondencia pares «cadena-valor» y genera como respuesta una página HTML. Esa estructura toma valores de un formulario, es decir, de una página *web* con campos que el usuario puede cumplimentar. El programa CGI puede, por ejemplo, consultar o modificar una base de datos y generar con el resultado una página HTML o un nuevo formulario.

Python y Perl son lenguajes especialmente adecuados para el diseño de interfaces *web*, pues presentan muchas facilidades para el manejo de cadenas y ficheros de texto. En Python tienes la librería *cgi* para dar soporte al desarrollo de aplicaciones *web*.

▶ 470 Diseña un programa que lea un fichero de texto en formato HTML y genere otro en el que se sustituyan todos los fragmentos de texto resaltados en negrita por el mismo texto resaltado en cursiva.

▶ 471 Las cabeceras (títulos de capítulos, secciones, subsecciones, etc.) de una página *web* se marcan encerrándolas entre `<Hn>` y `</Hn>`, donde `n` es un número entre 1 y 6 (la cabecera principal o de nivel 1 se encierra entre `<H1>` y `</H1>`). Escribe un programa para cada una de estas tareas sobre un fichero HTML:

- mostrar únicamente el texto de las cabeceras de nivel 1;
- mostrar el texto de todas las cabeceras, pero con sangrado, de modo que el texto de las cabeceras de nivel `n` aparezca dos espacios más a la derecha que el de las cabeceras de nivel `n-1`.

Un ejemplo de uso del segundo programa te ayudará a entender lo que se pide. Para el siguiente fichero HTML,

```
<HTML>
<BODY>
<H1>Un titular</H1>
<P>Texto en un párrafo.</P>
<P>Otro párrafo.</P>
<H1>Otro titular</H1>
<H2>Un subtítulo</H2>
<P>Y su texto.</P>
<H3>Un subsubtítulo</H3>
<H2>Otro subtítulo</H2>
<P>Y el suyo</P>
</BODY>
</HTML>
```

el programa mostrará por pantalla:

```
Un titular
Otro titular
  Un subtítulo
    Un subsubtítulo
  Otro subtítulo
```

▶ 472 Añade una opción a la agenda desarrollada en el apartado anterior para que genere un fichero **agenda.html** con un volcado de la agenda que podemos visualizar en un navegador *web*. El listado aparecerá ordenado alfabéticamente (por apellido), con una sección por cada letra del alfabeto y una línea por entrada. El apellido de cada persona aparecerá destacado en negrita.

El formato LaTeX

Para la publicación de documentos con acabado profesional (especialmente si usan fórmulas matemáticas) el formato estándar *de facto* es LaTeX. Existen numerosas herramientas gratuitas que trabajan con LaTeX. Este documento, por ejemplo, ha sido creado como fichero de texto en formato LaTeX y procesado con herramientas que permiten crear versiones imprimibles (ficheros PostScript), visualizables en pantalla (PDF) o en navegadores *web* (HTML). Si quieres saber qué aspecto tiene el LaTeX, este párrafo que estás leyendo ahora mismo se escribió así en un fichero de texto con extensión **tex**:

```
Para la publicación de documentos con acabado profesional
(especialmente si usan fórmulas matemáticas) el formato
estándar \emph{de facto} es \LaTeX.  Existen numerosas
herramientas gratuitas que trabajan con \LaTeX. Este
documento, por ejemplo, ha sido creado como fichero de texto
en formato \LaTeX y procesado con herramientas que permiten
crear versiones imprimibles (ficheros PostScript),
visualizables en pantalla (PDF) o en navegadores \emph{web}
(HTML).
Si quieres saber qué aspecto tiene el \LaTeX, este párrafo que
estás leyendo ahora mismo se escribió así en un fichero de
texto con extensión \texttt{tex}:
```

De acuerdo, parece mucho más incómodo que usar un procesador de textos como Microsoft Word (aunque sobre eso hay opiniones), pero LaTeX es gratis y te ofrece mayor control sobre lo que haces. Además, ¡puedes escribir tus propios programas Python que procesen ficheros LaTeX, haciendo mucho más potente el conjunto!

Ficheros de texto vs. doc

Los ficheros de texto se pueden generar con cualquier editor de texto, sí, pero algunas herramientas ofimáticas de uso común almacenan los documentos en otro formato. Trata de abrir con el Bloc de Notas o XEmacs un fichero de extensión **doc** generado por Microsoft Word y verás que resulta ilegible.

¿Por qué esas herramientas no escriben nuestro texto en un fichero de texto normal y corriente? La razón es que el texto plano, sin más, no contiene información de formato tipográfico, como qué texto va en un tipo mayor, o en cursiva, o a pie de página, etc. y los procesadores de texto necesitan codificar esta información de algún modo.

Hemos visto que ciertos formatos de texto (como HTML) permiten enriquecer el texto con ese tipo de información. Es cierto, pero el control sobre tipografía que ofrece HTML es limitado. Lo ideal sería disponer de un formato estándar claramente orientado a representar documentos con riqueza de elementos tipográficos y que permitiera, a la vez, una edición cómoda. Desgraciadamente, ese formato estándar no existe, así que cada programa desarrolla su propio formato de representación de documentos.

Lo grave es que, por razones de estrategia comercial, el formato de cada producto suele ser secreto y, consecuentemente, ilegible (está, en cierto modo, cifrado). Y no solo suele ser secreto: además suele ser deliberadamente incompatible con otras herramientas… ¡incluso con diferentes versiones del programa que generó el documento!

Si quieres compartir información con otras personas, procura no usar formatos secretos a menos que sea estrictamente necesario. Seguro que algún formato de texto como HTML es suficiente para la mayor parte de tus documentos.

HTML no es el único formato de texto. En los últimos años está ganando mucha aceptación el formato XML (de eXtended Mark-up Language). Más que un formato de texto, XML es un formato

que permite definir nuevos formatos. Con XML puedes crear un conjunto de marcas especiales para una aplicación y utilizar ese conjunto para codificar tus datos. Aquí tienes un ejemplo de fichero XML para representar una agenda:

```
<agenda>
  <entrada>
    <nombre>Pedro</nombre>
    <apellido>López</apellido>
    <telefono>964218772</telefono>
    <telefono>964218821</telefono>
    <telefono>964223741</telefono>
  </entrada>
  <entrada>
    <nombre>Antonio</nombre>
    <apellido>Gómez</apellido>
    <telefono>964112231</telefono>
  </entrada>
</agenda>
```

La ventaja de formatos como XML es que existen módulos que facilitan su lectura, interpretación y escritura. Con ellos bastaría con una orden para leer un fichero como el del ejemplo para obtener directamente una lista con dos entradas, cada una de las cuales es una lista con el nombre, apellido y teléfonos de una persona.

No todos los formatos son tan complejos como HTML o XML. De hecho, ya conoces un fichero con un formato muy sencillo: `/etc/passwd`. El formato de `/etc/passwd` consiste en una serie de líneas, cada una de las cuales es una serie de campos separados por dos puntos y que siguen un orden preciso (login, password, código de usuario, código de grupo, nombre del usuario, directorio principal y programa de órdenes).

▶ 473 Modifica el programa `agenda2.py` para que asuma un formato de `agenda.txt` similar al `/etc/passwd`. Cada línea contiene una entrada y cada entrada consta de 3 o más campos separados por dos puntos. El primer campo es el nombre, el segundo es el apellido y el tercero y posteriores corresponden a diferentes teléfonos de esa persona.

▶ 474 Un programa es, en el fondo, un fichero de texto con formato, aunque bastante complicado, por regla general. Cuando ejecuta un programa el intérprete está, valga la redundancia, interpretando su significado paso a paso. Vamos a diseñar nosotros mismos un intérprete para un pequeño lenguaje de programación. El lenguaje solo tiene tres variables llamadas `A`, `B` y `C`. Puedes asignar un valor a una variable con sentencias como las de este programa:

```
asigna A suma 3 y 7
asigna B resta A y 2
asigna C producto A y B
asigna A división A y 10
```

Si interpretas ese programa, `A` acaba valiendo 1, `B` acaba valiendo 8 y `C` acaba valiendo 80. La otra sentencia del lenguaje permite mostrar por pantalla el valor de una variable. Si añades al anterior programa estas otras sentencias:

```
muestra A
muestra B
```

obtendrás en pantalla una línea con el valor 1 y otra con el valor 8.

Diseña un programa que pida el nombre de un fichero de texto que contiene sentencias de nuestro lenguaje y muestre por pantalla el resultado de su ejecución. Si el programa encuentra una sentencia incorrectamente escrita (por ejemplo `muéstrame A`), se detendrá mostrando el número de línea en la que encontró el error.

▶ 475 Enriquece el intérprete del ejercicio anterior para que entienda la orden `si` *`valor condición valor`* `entonces línea` *`número`*. En ella, `valor` puede ser un número o una

variable y **condición** puede ser la palabra **igual** o la palabra **distinto**. La sentencia se interpreta como que si es cierta la condición, la siguiente línea a ejecutar es la que tiene el número **número**.

Si tu programa Python interpreta este programa:

```
asigna A suma 0 y 1
asigna B suma 0 y 1
muestra B
asigna B producto 2 y B
asigna A suma A y 1
si A distinto 8 entonces línea 3
```

en pantalla aparecerá

```
1
2
4
8
16
32
64
```